COMAL

for Commodore 64

Commodore

Consultants: UniComal ApS, Jels, Denmark
Editor: Jan Nymand, Commodore Data A/S
Cover and Illustrations: Fejltrøk, Silkeborg
Printed in Denmark by Werks Offset, Aarhus

The cover illustration suggests the evolution of COMAL for the Commodore 64 with its diversity of packages. The cover art and the drawings of the cartoon figures showing "the Superintendent's" encounters with COMAL are copyrighted by Fejltrøk, Christian XIII Vej, 8600 Silkeborg, Denmark.

Copyright

Assistance

If you have any comments concerning this COMAL manual or the programming language itself, please pass them along to your dealer. Commodore Data A/S has made every effort to assure that the contents of this manual are correct and complete and that the programming language itself functions as it should. Every error discovered by users will be corrected as soon as possible. Your help in this connection will be sincerely appreciated, not least by other COMAL users.

Responsibility

Neither Commodore Data A/S or any of this company's dealers or distributors give any guarantee expressed or implied concerning the COMAL computer language as described in this manual and tutorial. The language and documentation are sold "as is" with no claim being made as to its quality or suitability for specific tasks. The risk concerning the quality and performance of this product rests with the buyer. Should this product prove defective after purchase, it is the buyer (and neither the producer UniComal A/S, Commodore Data A/S, nor any distributor or dealer) who must take full responsibility for service, repair and any other costs accrued due to errors or defects. This is true even if the producer of the program has been given prior notice of the possible existence of such errors or defects.

Table of Contents


COMAL
comal for
64

Introduction

What is COMAL?

Welcome to the world of COMAL programming! Many feel that COMAL is close to being an ideal programming language for microcomputers, incorporating as it does the best features of Basic, Logo and Pascal. You are about to learn a programming language which offers, among other things, the following features:

* COMAL (COMmon Algorithmic Language) extends Basic, giving the language many of the powerful instructions of Pascal.
* COMAL retains the convenient operating environment of Basic, and many COMAL statements will be familiar to Basic users.
* COMAL for the Commodore 64 incorporates the easy to use turtle graphics which has made Logo famous.
* COMAL on the C-64 offers useful guidance when errors occur during program entry. The language contains structures for error handling during execution of programs.
* The language encourages structured programming with access to loop statements like:

  REPEAT - UNTIL
  WHILE - DO - ENDWHILE,

  flexible conditionals like
  IF - THEN - ELIF - ELSE - ENDIF
  CASE - OF - WHEN - OTHERWISE - ENDCASE

  and valuable building blocks like procedures and functions.

* COMAL for the Commodore 64 gives the user full access to the many special facilities which have made the C64 the most popular micro-computer in its class:

  high res color graphics
  sprites
  music
  joystick
  paddles, lightpen
  and much more...

The Origins of COMAL

COMAL originated in response to the needs of computer users in Denmark. Børge Christensen taught computer programming in the early 1970's to students at a small college in Tønder, near the German-Danish border. He found that the students often wrote terrible programs. They were hard to read, hard to de-bug and hard to maintain. Dr. Christensen consulted colleagues at the Institute of Computer Science at the Danish University of Århus. A researcher there by the name of Benedict Løfsted recommended that Christensen read the book, *Systematic Programming* by Niklaus Wirth.

Many readers will recognize Niklaus Wirth (of the Swiss Federal Institute of Technology in Zurich) as the father of the Pascal programming language. Inspired by Wirth's clear formulation of the principles of structured programming, Christensen contacted Benedict Løfsted. They agreed that the Basic language offered the user a very convenient operating environment. Basic was highly interactive. It allowed direct execution of instructions from the keyboard and required neither prior definition of variables nor the compilation process before a program could be run.

These features were ideal for a teaching environment. On the other hand Basic lacked the ability to use long, descriptive variable names and did not provide the elegant syntax of Pascal. If Basic could be augmented with these features, it would encourage the writing of clearer, better-structured programs. These men went to work with their colleagues to formulate requirements for the COMAL programming language. The language was developed and perfected during the 1970's. COMAL grew to maturity together with the personal microcomputer. The current version of COMAL 80, which you now own, is version 2.0. It is the product of standardization efforts by an international committee composed of representatives for users and industry. The COMAL kernel was agreed upon by this group. It is composed of all the COMAL instructions which must be common to all versions of the COMAL language. Special features, such as the use of graphics, music, sprites and other special features of the Commodore 64, are added as special packages. More about that later!

COMAL and Commodore

During the growth in popularity of the COMAL language, the Danish distributors of Commodore computers have played a leading role. With the advent of the inexpensive and popular microcomputer, in particular the Commodore 64, a group of young software enthusiasts, Jens Erik Jensen, Mogens Kjær, Helge Lassen and Lars Laursen formed a company, *UniComal ApS*. In cooperation with the Danish distributor and later with *Commodore Data A/S* they developed COMAL for the C-64.

When you have worked through the tutorial and written some of your own programs, we hope you will agree that the efforts of these pioneers have not been in vain!

Using this Tutorial

If you examine the Table of Contents, you will see that this manual begins with a chapter on setting up your computer and plugging in your COMAL cartridge. Next comes an easy to read, step-by-step introduction to COMAL programming. By the time you get to Chapter 3 we will assume that you have overcome the initial uncertainty (which everyone feels) when beginning with a new computer language.

Chapter 3 presents a systematic description of the most commonly used COMAL instructions. Here you will be presented with features for serious programming and begin to write your own COMAL programs.

Every programmer needs a good resource with information on the precise meaning of the most important facilities which are available in the language he uses. We have tried to provide this essential information - with examples - in a systematic form in Chapter 4, COMAL Overview. For those readers who require even more complete information on the definition of COMAL syntax, a comprehensive reference manual is available:

Len Lindsay, *COMAL Handbook* 1983.
Reston Publishing, 11480 Sunset Hills Road
Reston, VA 22090 USA (703) 437-8900
(also available from Prentice Hall International
66 Wood Lane End
Hemel Hempstead, Herts HP24RG, England
or from COMAL USERS' GROUP, 5501 Groveland Terrace,
Madison WI 53716 USA)

Note that the COMAL USERS' GROUP also puts out a newsletter. It contains many program examples and other useful information and is highly recommended. It is always a big advantage for the beginner to be in touch with more experienced users.

The *package concept* is one of those features which make COMAL for the Commodore 64 particularly powerful. When a special feature of your Commodore 64 (for example high resolution graphics) is to be used in a program, a package can be activated. When that feature is not needed, you don't activate the package. Turtle graphics are available, if you want to use them. Peripherals like joysticks, light pens, and paddles can be accessed with special packages of orders which extend the standard COMAL language. A complete description on the use of these packages is presented in Chapter 5.

Chapter 6 includes additional information on the handling of files in COMAL. This information will be particularly useful to those users who may wish to take advantage of COMAL to write programs for record keeping and data handling which require advanced facilities of the Commodore disk drive.

In Chapter 7 the use of peripheral equipment is covered. This includes the control ports to which you can attach joystick, paddles, or light pen, and the RS232 interface, IEEE interface, parallel port and other cartridges. This last item may be of particular interest to those users who may want to develop their own turn-key systems based on the Commodore 64.

Those of you with 16 fingers may want to get inside COMAL, learn about the use of Commodore memory and write your own machine language programs. This is also possible using COMAL. Read Chapter 8 to learn more about how this can be done.

This Tutorial concludes with a collection of information assembled in a series of Appendices. Here you will find the Commodore ASCII character codes, color codes for graphics, some tips on calculating with COMAL, use of the keyboard and the COMAL screen editor, use of strings, error messages and some useful programs and procedures. Finally there is an *Index* to help you find information quickly when you need it.

Work through the tutorial at your own pace. Be careful not to jump too far ahead before you're ready. Later on you should find this tutorial useful as a reference guide.

Happy programming!
Frank Bason & Leo Højsholt-Poulsen
Silkeborg, Denmark
January 1985

Chapter 1

Setting up Your Computer

Your Computer and Accessories

In order to use COMAL, you will require the following equipment:

* a Commodore 64 or 128 computer (or an SX-64 transportable)
* the COMAL programming language cartridge
* a video monitor or a television (color or B/W)

It is possible to run COMAL programs without an external storage medium - i.e. a disk drive or a tape unit. However, as you will soon be writing programs which you will want to save, it will be essential to have one of the following:

* a Datassette tape unit, or
* a Commodore 1541 disk drive.

Optional items of equipment for your Commodore 64 - nice to have but not essential - include:

* a Commodore printer or equivalent.
* an extra Commodore 1541 disk drive

When you begin to write longer programs, a printer is very useful to have. For serious programming you will need listings of your programs and printouts of your data. An extra disk drive is not an essential item. However, if you use your computer a great deal, a second drive will make copying programs and files a lot easier.

Figure 1.1 shows the connections on the rear and on the right side of your Commodore 64. Refer to this diagram to help connect the equipment you will be using.

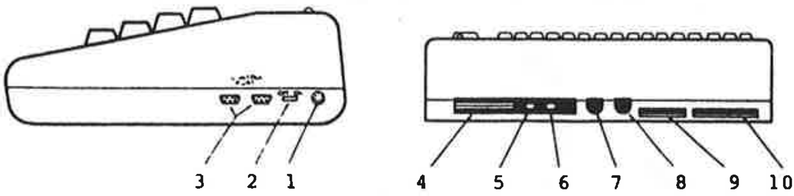


Figure 1.1: Accessories and peripheral devices are attached to your Commodore via the connectors on the rear and on the side of the computer. (1) power socket, (2) power switch, (3) game ports, (4) cartridge slot, (5) channel selector, (6) TV connector, (7) audio/video output, (8) serial port, (9) Datassette connector, (10) user port.

Your COMAL cartridge may also be used with the Commodore SX-64 portable version of the Commodore 64 computer. The SX-64 is illustrated in Figure 1.2. This unit includes both a color monitor and a disk drive unit. With a COMAL cartridge and the SX-64 you can skip ahead to the section on Installation of the COMAL Cartridge.

Figure 1.2: The Commodore SX-64 transportable computer is completely compatible with the Commodore 64. The SX-64 features a built-in color monitor and disk drive.

If you have access to a 1541 disk drive, attach it to the Commodore 64 via the 6-pole jack on the back panel (the same jack can be used for a printer).

If you have a printer as well as a disk drive, it can be connected to the second connector at the rear of the drive. You can use either one of the two connectors on the disk drive for the computer and for the extension cable to the printer.

If you are using a Datassette tape unit, attach it to the computer via the 12 pole edge connector (next to the User Port). Note that an ordinary tape recorder cannot be used.

You will find detailed information on the use of these accessories in your Commodore 64 manual, and in the disk drive, Datassette and printer manuals.

A typical set-up will look like the illustration in Figure 1.3. The system shown includes a single disk drive, a printer and a portable TV used as a display.

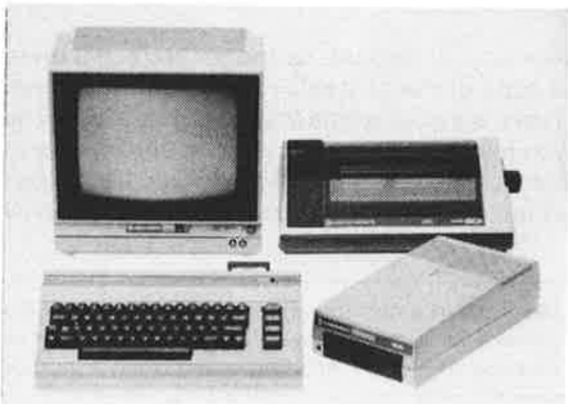


Figure 1.3: An ideal setup for learning and using the COMAL programming language includes a Commodore 64 computer equipped with a printer, 1541 disk drive and a color TV or monitor.

Don't turn anything on yet. We will have to install the COMAL cartridge before continuing!

Installing Your COMAL Cartridge

Your COMAL language cartridge is shown in Figure 1.4. It must be installed in the cartridge slot at the rear of your computer, if you use the Commodore 64 or 128. If you have an SX-64, then the cartridge slot is on top of the machine on the right hand side.

Figure 1.4: Your COMAL cartridge allows you to expand the power of your Commodore 64 without using additional memory. It fits into the cartridge slot at the rear of the Commodore 64 (or the top of an SX-64).

Take a closer look at your COMAL cartridge. Note that there is a row of contacts on the edge of the printed circuit board which protrudes from the cartridge. There is a label on the *front* of the cartridge. This must face upward when you insert the COMAL cartridge horizontally into the cartridge slot of the Commodore 64. (The label will be towards the front, when you insert the cartridge into the cartridge slot of an SX-64.)

WARNING: Never insert a cartridge into your Commodore 64 or SX-64 (and never remove it) with the power turned on. The power to all peripherals must be OFF when inserting or removing a cartridge!

When you are sure that the cartridge edge connector is properly aligned with the slot in the computer, push the cartridge firmly into place using a gentle rocking motion.

Connecting the TV or Monitor

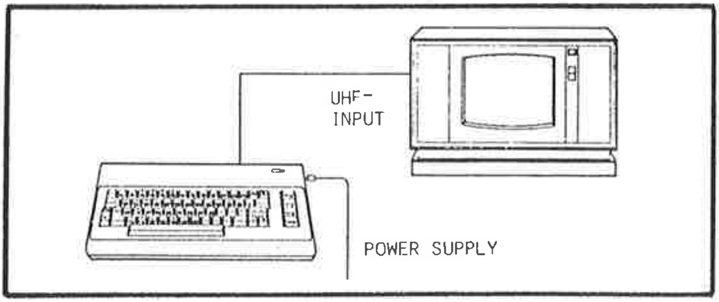


Figure 1.5: The C-64 can be connected to the input of a standard TV receiver.

Connecting the TV or Monitor

Your Commodore 64 is supplied with the following display outputs:

* color monitor signals (audio, composite video and luminance)
* a modulated standard color TV signal

The output jacks for these signals are shown in Figure 1.1.

Because the SX-64 has its own color monitor, the following discussion will only apply to the Commodore 64. If you will be using the SX-64, you can proceed directly to the next section on running the demonstration program.

A color monitor is the most desirable choice of display for your Commodore 64, because it will give you the sharpest image. If you have a Commodore monitor, just attach one end of the connector cable supplied with the monitor to the 8-pole connector on the rear panel of the Commodore 64. Plug the connectors at the other end of the cable into the three phono jacks on the rear panel of the monitor. If you will be using a different type of monitor, your dealer will be able to assist you to find the proper cable.

A TV connector cable is supplied with your Commodore 64 for those users who will be using a color (or B/W) television set for their display. If you will be using a television set, insert the phono plug end of the cable into the phono jack on the rear panel of your Commodore 64, and plug the other end into the antenna input jack on your television receiver.

You must also connect the Commodore transformer to your computer. The cable from the power supply is inserted on the right hand side of your computer (towards the rear, right next to the power switch).

When all connections have been properly made and all shipping protection has been removed from your disk drive and printer, you are ready to turn on your equipment. To do this both the switch on the power supply as well as the switch on the right side of the computer must be turned on.

found demoprogram

After a minute or so the cursor will begin blinking again, indicating that the loading operation is completed. (You can interrupt the read-in by pressing the Commdore key <**C=**>.) You are now ready to run the demonstration program.

If you have difficulty loading the demonstration program, you can try the following:

* Turn off the power to the computer and the Datassette, and check again that the tape recorder is connected correctly. Is the cable intact and plugged all the way in?
* Be sure you are using the correct tape and that it has been rewound all the way back to the beginning (all the tape should be on the left hand reel).
* When you have checked the above points, apply power to start COMAL up again. Then repeat the read-in procedure.
* If you still have difficulty, contact you dealer for assistance.

Using a Disk Drive:
If you have a disk drive, insert the COMAL Demonstration Diskette. The label should face upward and be on the edge facing you when the diskette is inserted (see Figure 1.6).

Figure 1.6: Handle the diskette carefully. Open the drive door, and insert the diskette into the drive as shown. Slowly push the diskette all the way into the slot. When the diskette is in place, close the drive door until you hear it click into place.

Now type:

load "demoprogram"

and press <**RETURN**>. This instruction will transfer a copy of the program from the diskette to your computer's memory. The activity indicator on the drive should light up, and the drive will operate for a few seconds.

Whichever means you have used to load the demonstration program, you can now type **run** and press <**RETURN**>. Then sit back, relax and enjoy the show! Be sure your TV or monitor sound volume is turned up slightly so music and sound effects can be heard.

You can interrupt the progra... <**RUN/STOP**> key.

Be sure to remove the demo diskette and store it in a safe place before proceding with the next section of this chapter.

If you follow the tutorial in the coming chapters, you will soon be able to adapt the powerful features of your Commodore 64 with the COMAL programming language - high resolution color graphics, sprites, sound and more - for use in your own programs.

Preparing a Storage Diskette

Before we proceed to the introductory tutorial in Chapter 2, let's get a blank diskette ready for storing your programs. This process is called *formatting* the diskette. Datassette users won't need to format tapes - that is not necessary. But tape users may want to read this section anyway to learn more about diskettes and how they are used.

You should interrupt the COMAL demo program so that you can enter commands from the keyboard. Press <**RUN/STOP**>, if you haven't already done so.

- Be sure that the demo-diskette has been removed and stored away.
- Take a diskette which is unused (or which can be erased). Be sure that the *write protection notch* is uncovered (see Figure 1.6). Cove-

Tune the channel selector of your color TV to find the signal. Adjust the TV receiver for the sharpest possible picture.

If all has gone well, the following message should be present on your display screen:

$$$ Commodore-64 COMAL 80 rev 2.01 $$$
(C) 1984 by UniComal & Commodore
30714 bytes free.

and the blinking cursor will appear 3 lines below the message. If the sound is turned up on your TV or monitor, you will also hear a signal, indicating that COMAL is ready to go.

Should any problem arise at this point, **turn off your equipment** at once. Check the setup procedure once again. Be sure that the COMAL cartridge is inserted correctly and is firmly seated in its socket. Check all cables and be sure that there is power at the electrical socket. Check your Commodore 64 Instruction Manual. If problems persist, contact your Commodore dealer for help.

The Commodore Keyboard

If you are not familiar with the Commodore keyboard, then type anything at all, just to get used to it. Try out the <**SHIFT**> and <**SHIFT LOCK**> keys. If you should make a typing error, be sure that the <**SHIFT LOCK**> key is disengaged, then press the "insert or delete key" marked <**INST/DEL**> at the upper right hand side of the keyboard to delete the character just to the left of the cursor.

You can also move the cursor around the screen using the cursor control keys (next to the right hand <**SHIFT**> key). If the <**SHIFT**> key is depressed and you press <**INST/DEL**> then extra spaces appear, allowing you to make insertions. Try out the <**CLR/HOME**> key with and without the <**SHIFT**> key engaged to learn what it does.

If you have a black/white TV receiver or monitor, hold down the <**CTRL**> key at the left of the keyboard. Press the letter **W** at the same time. Doing this will change the screen and cursor colors, making the screen easier for you to read. If you are using a color TV or monitor, try <**CTRL**> **V** for a dark blue background and white text. More on color changes later on!

You might try pressing the Commodore Key <**C=**> (on the left hand side of the keyboard) and the <**SHIFT**> key at the same time. When you do this you will "toggle" the display back and forth between *capitals and small letters* and *capitals* and *graphics characters.* Be sure you have selected *capitals and small letters*.

Check:	Press the keys	<A>	<S>	<D>
	The computer prints	**a**	**s**	**d**
	Press the same keys again while holding down <**SHIFT**>.			
	The computer prints	**A**	**S**	**D**

For the time being, the features described here will be adequate for proceeding with this tutorial. You will learn about additional facilities, as we go along. A complete description of the keyboard and the many features of the COMAL screen editor is available in Appendix D.

Running the Demonstration Program

If your Datassette tape storage unit or your disk drive is connected properly, you are ready to run programs. Please read the instructions which apply to you:

Using the Datassette:
If you are using a Datassette unit for program storage, insert the COMAL Demonstration Tape and type:

load "cs:demoprogram"

then press the key marked <**RETURN**>.

Note that if you intend to use the Datassette from now on, you can make it the default unit by typing in the command:

unit "cs:" <**RETURN**>

Note that a word like RETURN printed within brackets < > means to press the key with that name instead of spelling out the entire word on the keyboard.

The C-64 responds by printing **press play on tape** on the screen. Be sure that the tape has been rewound to the beginning then start the tape by pressing the **PLAY** button on the Datassette. The computer responds:

ok
searching for demoprogram

The screen will go blank for a moment. When the program has been located, the following message will be displayed:

ring this notch with a piece of tape prevents formatting or changing the contents of the disk by saving new files.

* Insert the diskette correctly into the disk drive, and close the drive door, so it clicks into place.
* Now type the following instruction:

pass "n0:my diskette,XX"

When you press <**RETURN**> the disk drive will begin to operate and continue for about 2 minutes. The disk activity light will go out, when the formatting process is finished. You can now use this diskette for storing your programs and files.

A few remarks about the command which you just issued from the keyboard: **pass** indicates to COMAL that the subsequent text should be passed to the disk drive. The letter **n** is the code for formatting a new diskette, and **0** means that it should be done on the first of your drives (in case you have more than one). You are free to choose the <**diskette name**> - up to 16 characters. This name (**my diskette** in this example) will appear as a heading whenever you catalogue your disk (more about this in Chapter 2). The last entry **XX** may be any two characters. It serves as a diskette identifier code.

Review

Your equipment should now be set up and ready to use. You have mounted the COMAL cartridge, powered up, and familiarized yourself with the keyboard. You have also read in a demonstration program and run it to check out your system.

The program has given you a preview of the impressive potential of this programming language. Finally, if you will be using a disk drive, you have formatted a diskette which can be used for storage of programs as you work through the tutorial chapters which follow.

Chapter 2

Let's get started!

Why learn to program?

The computer is a tool for handling information. Properly programmed, your Commodore 64 can do calculations, manipulate text, sort data, collect data, control machines, create images, make sound, and much more. The heart of the computer is the now well-known component called the *microprocessor* If it is connected to sufficient memory and a means of getting data in and reading data out, we have a *microcomputer*.

The elementary operations which the microprocessor performs on individual *bytes* of data are very simple. Just adding two numbers like 2543 and 9320 together may require the microcomputer to perform hundreds of simple operations. Yet because each operation only takes a millionth of a second, the job is done in a thousandth of a second!

When you program a computer, it is possible (but by no means necessary!) to work with the fundamental binary numbers used by the processor. Your Commodore 64 computer uses a *6510 chip*. You can use *assembler language* if you want to program it directly. More on this subject is available in Chapter 8.

To make life easier for programmers, *higher level languages* have evolved which allow the use of very simple instructions to accomplish a large number of elementary processor operations. A statement like:

print 2543 + 9320

is an example of a high level instruction.

This statement causes the two numbers to be added together and printed on an *output device*, say a display screen.

An ideal computer language allows the programmer to group sets of instructions together to perform more complex tasks and to give them a new name. For example, it would be nice to have an instruction like

interest(12535,8)

which could compute the interest accumulated by an investment of 12535 dollars (or pounds) during an eight year period. While everyone using a computer language will want to add numbers, not everyone will require this particular procedure. So the ideal language will include a large number of useful standard procedures and make it easy for the programmer to construct his own special ones.

COMAL is such a language. It is a *procedure oriented* language which includes many clear and useful elementary instructions for custom building your own procedures. Your procedures may be so useful that they themselves can be used again in other programs or in other procedures which handle larger tasks. The COMAL *operating enviroment* makes this easy and convenient to do. When you have learned the COMAL language, you will have a very powerful tool indeed to help you solve a wide range of problems.

Learning a powerful programming language is an adventure. Adventures can be fun and exciting. But no adventure worthy of the name is without challenges and pitfalls. The ability to write your own programs will come only with practice, persistence, curiousity and patience. You have begun an adventure and must be prepared to go through periods of trial and testing before you become a seasoned programmer.

Programming is not just for solving serious professional problems. It can be fun, too! Just ask any programmer after a late evening getting his own game program to work. The thrill of bringing a program to life after carefully building it up out of its component parts can be compared to other highly creative activities. (Don't ask the programmer about this before he or she has found the last bug and gotten the program to run!) Programming can be used for so many purposes that it is impossible to provide a complete list. Here are just a few; you can probably think of many more. Properly programmed, your computer can:

* play a game with graphics to help children learn
* help teach music by showing notes and playing sounds
* prepare an expense summary and compare it with your budget
* keep sales records for a small business
* record and display weather records
* make measurements in the lab or on a production line
* prepare an income tax return and print it out
* help plan and administer a construction project

* compute the heat losses from a building
* provide motivating teaching aids to help students learn

A great deal of programming today has to do with *games*. Since the earliest days of programming, programmers have loved to use their machines for "play". (Rumor has it that in the late 1960's **Star Trek** was the most popular program at many university computing centers.) When computer time cost hundreds of dollars an hour, it was a luxury few could enjoy. Today microcomputer time costs only a few cents per day, so game programs have proliferated as never before. If you want to play computer games or write some yourself, then welcome to COMAL. It is a fast language with excellent color graphics, sprites and sound effects. The possibilities for game programs are endless.

Of the many program types, perhaps *simulations* are the most fascinating. You can become the pilot of a World War I fighter plane in hot pursuit of enemy planes. Change the program parameters, and you are piloting a 747 jet to a landing at Paris, London or New York. Or simulate Charles Lindberg's aircraft, the Spirit of Saint Louis on the first non-stop New York to Paris flight. Even the flight of the Space Shuttle or the Concorde can be effectively simulated using a microcomputer. With color graphics and a joystick, such simulations can be strikingly realistic.

But simulations can be much more than this. They can be effective tools for learning - both for students and for professionals. With simulation programs you can, among many other possibilities, examine:

* the financial decisions of a business
* the operation of a solar heating system,
* the operation of a nuclear reactor,
* the motion of charged particles in electric and magnetic fields,
* the orbiting of a satellite,
* or the flight of a rocket.

Again, for those who undertake the adventure of learning to program the possibilities are almost unlimited. Limited in fact only by your imagination and your ability to use the tools which you now own: your Commodore 64 computer and the COMAL programming language. Let's learn more about how to use them!

Direct Execution of COMAL Commands

Your computer should have the COMAL cartridge installed and should be turned on. When you do this the following message should appear on the screen:

$$$ Commodore-64 COMAL 80 rev 2.01 $$$
(C) 1984 by UniComal & Commodore
30714 bytes free.

If this message is on your screen, then you are ready to proceed...
For a starter, try pressing <**CTRL**>**-V** to change the screen colors to a pleasing blue with a white cursor and text. If you're using a B/W display, try <**CTRL**>**-W** for a grey background and black text.

IF YOU MAKE A TYPING ERROR: You can delete the character just to the left of the cursor by pressing the <**INST/DEL**> key at the upper right of your keyboard. (The <**SHIFT LOCK**> key must not be depressed when you do this!) For a complete description of the use of the keyboard and a run-down on the many editing facilities available with COMAL, see Appendix D.

The simplest way to start using COMAL is to type some direct instructions from the keyboard. Try typing:

print "hello"

When you press <**RETURN**> the word **hello** should be printed on the next line on your display screen.

It is important to understand that the computer first *processes* your direct instructions when you have pressed <**RETURN**> giving in effect an instruction to process the current command line.

Note that instructions may be entered in either lower case or upper case. (You toggle the display screen between upper case/graphics and lower case/upper case by pressing the <**C=**> and the <**SHIFT**> keys at the same time.)

We will assume in this tutorial, unless otherwise stated, that the *lower case/upper case mode* has been selected.

You can also do calculations using direct instructions. Try the following instruction, being careful NOT to press the <**SHIFT**> key when typing the + sign:

print 217+305

After pressing <**RETURN**> you will see the computer print the number **522** on the next line.

You can also mix text and numbers in a PRINT instruction as in the following example:

print "sum =",217+305

After you have entered the instruction by pressing <**RETURN**> the computer will print:

sum=522

Notice that if you give no other instructions, the text and the number will not be separated by any spaces when they are printed. This can be changed by using a *semicolon*; If a semicolon is used as a separator, then a blank space will be printed to the right of each text segment or number.

You can also arrange the placement of your text and numbers on the screen using the ZONE instruction. Type:

zone 10

We want to repeat the same instruction used earlier. For a work-saver try this little trick: Depress the <**SHIFT**> key and press the cursor up-down key (just below <**RETURN**>). Move the cursor up the screen until it is blinking on the line:

print "sum =",217+305

Release the <**SHIFT**> key and press <**RETURN**> Your instruction will be executed again. But this time there will be 10 spaces between the start of the text to the first digit of the number. The ZONE instruction is used to specify the width of the printing columns when text or numbers are separated by *commas*. The default condition ZONE 0 is set when you start up your system.

You may want to do some experimenting with ZONE and PRINT instructions before moving on in this tutorial. This is easy to do by using the cursor keys to move up and down on the screen. Notice that you needn't be at the end of a line on the screen for the instruction to be executed. Notice also that if extraneous text is on the same line, it will be interpreted together with the instruction you want to execute, and an error message will result. You can either delete the extra text (<**CTRL**>-**K** will delete everything from the cursor position to the end of the line), or you can write your instruction on an empty line to avoid this error. You can also completely erase the screen by executing the instruction PAGE or by holding down the <**SHIFT**> key while pressing the <**CLEAR-/HOME**> key.

COMAL has many other facilities for handling text and numbers. We'll be looking at these in much greater depth later on. Before we proceed to write programs, let's take a quick look at how to use the high-resolution graphics screen.

A Quick Look at Turtle Graphics

Your Commodore 64 is almost ready to do **turtle graphics** as soon as you power up. Just press <**f3**> to enhance COMAL with the instructions in the *turtle graphics package*. When you press <**f3**> the words **USE turtle** will appear on the screen. Then the appearance of your screen will change. A small arrowhead will appear in the middle of the screen, and the words USE turtle will now be at the top of the screen with the cursor blinking on the next line. You are now looking at the *split screen* with four lines of text visible at the top. Pressing <**f1**> will bring you back to the *text screen*. If you depress <**f5**> you will be looking at the *graphics screen*. The entire screen can be used for graphics, but you will not be able to see your instructions as you type them in. Now press <**f3**> again to get back to the split screen.

Notice that by means of the USE instruction you have extended the COMAL language with a set of extra instructions, called a *package*. As you will learn, many other packages are available in your COMAL cartridge. Much more about packages in Chapter 5!

If you should want to remove the COMAL extensions invoked by the instruction USE, you can type:

discard <**RETURN**>

This will remove ALL packages from program memory. (You cannot remove packages selectively.) Typing **new** will delete your program and deactivate all packages as well.

Let's see how the *turtle* (also called the *graphics cursor* or the *pen* represented by the arrowhead can move around the screen and draw. We'll use direct commands now but we will write a complete program later on in this tutorial.

Turtle graphics instructions are so straigtforward that you can learn how they work just by trying them out. Try typing:

forward(50) <**RETURN**>
right(90) <**RETURN**>

Type the same instructions again. You should have a square halfway finished on your screen. Use these turtle graphics commands again as needed to complete the square. The turtle should end up pointing upwards again.

Now try the following instructions (remembering to press <**RETURN**> after each) and observe what effect they have on the turtle and the drawing:

penup
back(50)
pendown
forward(50)

Notice that if your experimentation brings you too far in any direction, the turtle will show up at the other side of the screen.

Type **home** to bring the turtle back to the center again, then type **clearscreen** to erase the screen. You can also type **home;cs** on one line to accomplish this.

Now try:

left(90)
forward(50)
setheading(45)
forward(70)

What does each instruction do? Do some experimenting yourself to understand how to move the turtle and make it draw. You might want to try the following sequence:

for side=1 to 4 do forward(50);left(90)

This example illustrates a unique feature of COMAL: A *sequence* of procedure calls or assignments separated by a semicolon can be executed directly from the keyboard!

To illustrate how COMAL actively assists you as you type in instructions (if you haven't already noticed this), try making intentional errors while typing in the previous command:

type: **for** <**RETURN**>

Note the computer's response.

type: **for =** <**RETURN**>

Note the response.

type: **for i =** <**RETURN**>

Note response, etc.

Another aid provided by Commodore COMAL is that the error messages are removed from the screen as soon as you have corrected the error and pressed <**RETURN**> or moved the cursor to another line.

Note what each of the following instructions does:

hideturtle
showturtle

If you have a color display you can also experiment with

background(<number>)
pencolor(<number>)

where <**number**> is a *color code*. See Appendix B for a list of the *graphics color codes.*

The table which follows shows turtle graphics instructions with a short form for each and a brief description. When you give the instruction **use turtle** from the keyboard or in a program, all these instructions as well as all the commands in the *graphics package* become available for you to use.

TURTLE ORDER	SHORT FORM	DESCRIPTION
back(L)	bk(L)	move L units backwards
forward(L)	fd(L)	move L units forward
background(C)	bg(C)	background color set to C
clearscreen	cs	clears the graphics screen
home		turtle to screen center
hideturtle	ht	conceals the drawing cursor
showturtle	st	shows the drawing cursor
pencolor(C)	pc(C)	sets the drawing color to C
pendown	pd	cursor leaves a trace
penup	pu	cursor leaves no trace
left(D)	lt(D)	cursor turns D degrees left
right(D)	rt(D)	cursor turns D degrees right
setheading(H)	seth(H)	cursor points to heading H H=0 is up, 90 is right,etc.

Make careful note of these instructions. We will be using them again in the program examples which follow.

What is a program?

In order for a machine or a computer to do a job, it has to be "told" how to do it. In contrast to a human being who can base his actions on skills and experience, the machine must be given very precise instructions. Nothing must be taken for granted. In practice this means writing down a list of orders, each of which can be interpreted by the computer, describing in detail the job to be performed.

This could be a very tedious task indeed, if we were obliged to give details on how to, say, "add two numbers together" each time it had to be done. This is of course not necessary. When the computer has been instructed on how to interpret the instruction **PRINT x + y** where **x** and **y** are any pair of numbers, it can add any two numbers at all (within certain very wide limits - see Appendix C). The same is true of other operations we expect the computer to do. A few of the most commonly used operations:

* adding, subtracting, multiplying and dividing numbers
* printing numbers and text
* drawing a line from point to point
* making a choice of two paths to follow
* repeating operations a certain number of times,
* selecting different tasks when certain conditions are met,

are defined in a *computer language* which is relatively easy for human operators to use. COMAL is special, because this language is regarded by many as a particularly clear, powerful and flexible language.

Let's try writing a COMAL program to illustrate some of these ideas.

Suppose we want to draw a square on the display screen of the computer. Even with no prior knowledge of programming, we could write down a list of the tasks to be accomplished, using everyday English:

* Get the computer ready to use the screen for graphics.
* Describe how far to move and how much to turn to draw a side of the square.
* Repeat the above step four times to complete the square.

Being a bit more specific, we could express this by writing the following instructions. We intend to draw a square 75 "units" on a side starting at the center of the screen. We want the sides of the square to be parallel with the edges of the screen:

* Set the turtle graphics mode.
* Move the pen forward 75 units, and turn right 90 degrees.
* Move forward 75 units again, and turn right 90 degrees.
* Move forward 75, and turn right 90.
* Move forward 75; turn right 90.

When all this is accomplished, we should have a square on the screen with the drawing cursor back in its original position. It is usually good programming practice to leave the turtle at the end of an instruction sequence in the same *state* as it was when the sequence began. This idea is particularly important when you begin to write COMAL procedures. It makes things easier when you want to build a program up using "modules" or "building blocks" which must work together to do a job.

Let's see how the actual COMAL program would look. Note that it may not be clear at once why certain things are done. As you progress with this tutorial you will be presented with more thorough explanations to reveal most of these mysteries!

First be sure you are using the *text screen* (press <**f1**> if you have been using graphics). Be sure that no other COMAL program is in memory (type **new** <**RETURN**>). You will probably want to clear the screen and move the cursor to the top left side of the screen. Press the <**SHIFT**> key and the <**CLR/HOME**> key at the same time to do this.

If you have trouble getting your computer into text mode with the screen cleared, there is one sure-fire way of getting things straightened out. Depress the <**RUN/STOP**> key and hold it down while pressing the <**RESTORE**> key. This action will initialize things without loosing your program.

Of course you can always turn off the computer power switch, wait a few seconds, and turn it on again. You should be back in COMAL with the greeting message on the screen, ready to go, but this solution will erase your program.

When you prepare a program, the instructions you prepare are not executed right away. They are stored in memory and only executed when the program is *run*. You will find that line numbers are not important in COMAL except as an aid when entering and editing a program. In fact you will be able to completely ignore line numbers when your program is completed.

To make program entry easier, press <**f4**> to get automatic line numbering. (You get this by pressing <**SHIFT**> and the <**f3**> key.) COMAL responds with **AUTO** Press <**RETURN**> and automatic line numbering will be engaged.

The computer should be ready to accept instruction number **0010** Note that it is usually wisest to number instructions with intervals of 10, so that there will be room to make insertions in case you discover later on that an instruction has been left out.

To get rid of automatic line numbering or to change it, just press <**RUN STOP**> instead of entering a new line. If you then type **auto** or press <**f4**> again, you will be back to automatic numbering at the line you left. You can add one or two numbers to the AUTO command to change the starting line and the line number interval. If you type **auto,5** <**RETURN**> the line number interval will be 5 (the line numbers will continue from where you were). If you type **auto 100,5** then line numbering will start at line 100 with a line number interval of 5.

Recalling our list of plain English tasks to be performed, we can start with the COMAL instructions which must be used to prepare the screen for turtle graphics:

```
0010  use turtle
```

Press <**RETURN**> after each instruction line (although multiple instructions on the same line separated by ; are sometimes allowed, usually only one instruction per line is recommended). As you enter program lines, COMAL prints the next program line number, ready for your next instruction. Type as follows to continue with our sample program. Use the cursor keys and the <**INST/DEL**> key as needed to correct any typing errors. Feel free to use the abbreviated instructions if you prefer.

```
0020  splitscreen
0030  forward(75)
0040  right(90)
0050  forward(75)
0060  right(90)
0070  forward(75)
0080  right(90)
0090  forward(75)
0100  right(90)
0110  while key$=chr$(0) do null
```

After your experience with the turtle in the last section these instructions should be easy to understand except perhaps for the instruction in line number 110. We want to keep the graphics screen visible after drawing the square. When a COMAL program ends while using graphics, control returns automatically to the textscreen screen, so that you can see your instructions as you type. Line 110 makes the graphics screen remain completely visible until you press any key. When **key$** no longer equals the default value **chr$(0)** the program will continue beyong line 110. When the program proceeds beyond this line, there are no more instructions, so the program will stop.

Try running the program. First press <**RUN/STOP**> to get out of AUTO mode. Then type in **run** When you press <**RETURN**> your program will be carried out step by step. This process is called *executing* a program.

You can save a little effort if you want by pressing <**f7**> instead of typing in **run**

Press <**f1**> to return to the text screen. Change the program and run it again to see what happens. Try different lengths and different angles to make other figures. When you have finished experimenting, we'll go on to look at some additional COMAL instructions.

Notice that pressing the <**f3**>-key activates graphics mode while disabling the default function of the key. Pressing <**f3**> again after say a program stop, does not re-initialize turtle-graphics. Press <**CTRL-u**> to reactivate <**f3**>.

Repeating Instructions

After working with the sample program to draw the square - and perhaps after trying to draw pentagons and octagons - you may wish it were possible to repeat a given set of instructions which you want to use repeatedly. It is indeed possible. This programming structure is called a *loop block* and is one of the most important concepts in programming.

There is an easier way to draw a square. Erase program memory using **new** <**RETURN**> and try the following program:

```
0010 // program: SQUARE
0020 // by: <your name>
0030 use turtle
0040 splitscreen
0050 for sides:=1 to 4 do
0060 forward(75)
0070 right(90)
0080 endfor
0090 while key$=chr$(0) do null
0100 end // of program
```

Press <**RUN/STOP**> to stop auto-numbering then write **list** to do a listing of your program. It should look like this:

```
0010 // program: SQUARE
0020 // by: <your name>
0030 USE turtle
```

```
0040 splitscreen
0050 FOR sides:= 1 TO 4 DO
0060   forward(75)
0070   right(90)
0080 ENDFOR
0090 WHILE KEY$=CHR$(0) DO NULL
0100 END // of program
```

As you can see, it is possible to add titles, bylines and other comments to your programs. Just precede them with a //. Such statements are not executed, but they will appear in your listings. They can also be added after COMAL instructions in a program line, as in line 100. Notice how COMAL indents lines 60-70 in the listing to make the structure of the program clearer. The FOR-ENDFOR construction (50-80) causes lines 60-70 to be repeated four times. Also keywords are capitalized in the second listing.

Now SCAN your program by pressing <**f8**> or issue the direct instruction **scan**. (This process will also check through your program for errors in structure and define any procedures in the program.)

Another LIST will show that the variabel name **sides** has been included after ENDFOR in the program listing.

You have seen how COMAL edits your programs to provide a clearer listing. From now on in this tutorial, we will show programs in their final, edited form. It will, however, probably be easiest for you to continue typing the programs in lower case. Let COMAL do the extra work of providing a nice listing for you!

Try running the program square . Press any key to stop the program, then press <f1> to return to the full text screen. No let us try som changes to see what happens. Can you alter the program to cause it to draw a hexagon (6 sides) or an octagon (8 sides)? When instructions are to be repeated many times, the FOR-ENDFOR construction becomes particularly useful. Can you adapt the program, so the turtle draws a figure which is close to being a circle?

You may have noticed that in order to complete a polygon and end up facing in the same direction as when it started, the turtle must turn a total of 360 degrees. (Those of you who are familiar with the computer language Logo, which also uses turtle graphics, may recognize this principle as the *Total Turtle Trip Theorem*.) So to draw a regular polygon with **number** sides, the turtle must turn **360/number** degrees at each vertex.

It is of course possible to adapt this program so that it will draw a polygon with any number of sides we choose. To do this we will have to indicate the number of sides desired and the length of a side by means of INPUT statements. Erase program memory (**new <RETURN>**, and try entering the following program:

```
0010 // program: polygon
0020 // by:  <your name>
0030 PAGE // clear the screen
0040 INPUT "How many sides? ": number
0050 INPUT "Length of each? ": length
0060 USE turtle
0070 splitscreen
0080 FOR sides:=1 TO number DO
0090   forward(length)
0100   right(360/number)
0110 ENDFOR sides
0120 WHILE KEY$=CHR$(0) DO NULL
0130 END // of program
```

Note that the program is shown here as it would be listed. You can enter the program in lower case and without indentation, if you wish. Run it to be sure it works as expected.

COMAL Procedures

Procedures are modules or building blocks which you can create to make your programming easier. There is a line in the program **polygon** which lends itself to being redone as a procedure. You can make your program easier to read and easier to understand by creating a procedure. This technique becomes very important when you begin to write longer programs!

Notice that the use of *line numbers* in COMAL is quite different from their use in other line-oriented languages such as BASIC. In this respect COMAL is much more akin to Pascal. Use the RENUM instruction often to "clean up" your program. Because few COMAL instructions ever refer to a line number, you can pay much less attention to them. In general it is probably best to group your program instructions into three sections:

beginning
program name, date, comments,
dimensioning of variables,
setup of packages, etc.

middle
the main program sequence
consisting mainly of
procedure calls

end
collection of your procedures
called by the main program

Take a look at your program. Consider statement number 120:

```
0120 WHILE KEY$=CHR$(0) DO NULL
```

used here as in the program **square** to keep the graphics screen visible until any key is pressed. It could be made into a procedure to keep it from cluttering up the main program:

```
0140
0150 PROC wait'key
0160   WHILE KEY$=CHR$(0) DO NULL
0170 ENDPROC wait'key
0180
```

Notice here that we have called the procedure **wait'key** The *apostrophe* ' is needed to bind the two words describing the procedure together into one continuous string of characters with no blanks. If this is not done, COMAL will only interpret the letters before the first blank as the procedure name, and an error message will result when COMAL tries to execute the procedure.

Add this procedure to your program, and replace line 120 by:

```
0120 wait'key
```

Now list the procedure (a little trick: use <f6> <**RETURN**> to do this). Notice the following features of the COMAL listing:

* The LIST instruction *indents instructions* in the procedure, setting the procedure apart and making the program listing easier to read.
* The procedure must be terminated by ENDPROC. If the program has been SCANned or RUN, then COMAL includes the *name of the procedure* in the ENDPROC instruction, if you have not already done so.

* The *blank lines* in lines 140 and 180 are not required. They are included to cause this procedure to be separated more clearly from others when the program is listed.

The program **polygon** could be improved further by creating a procedure out of the statements which actually draw the polygon.

The polygon procedure might be typed in like this:

```
1200 proc polygon(number,length)
1210 for sides:=1 to number
1220 forward(length)
1230 right(360/number)
1240 endfor
1250 endproc
1260
```

When you SCAN and then LIST the procedure, it should appear as follows:

```
1200 PROC polygon(number,length)
1210    FOR sides:=1 TO number DO
1220      forward(length)
1230      right(360/number)
1240    ENDFOR sides
1250 ENDPROC polygon
1260
```

There are a few things you should notice about the listing:

* The *procedure name* is followed by two *variable names* (**number, length**), indicating that the procedure will require values for the **number** of sides and the **length** of each side. A procedure need not have any variable list after its name (like the procedure **wait'key**. It can have one, two or more indicated, as shown here.

Again we must *call* the procedure before it can be executed. The original program must be changed, so it looks like this when RENUMbered and LISTed:

```
0010 // program: polygon
0020 // by: <your name>
0030 PAGE
0040 USE turtle
0050 splitscreen
0060 INPUT "How many sides? ": number
0065 INPUT "Length of each? ": length
0070
0080 // MAIN PROGRAM
0090 polygon(number,length)
0100 wait'key
0110 END // of MAIN PROGRAM
0120
0130 PROC wait'key
0140    WHILE KEY$=CHR$(0) DO NULL
```

```
0150 ENDPROC wait'key
0160
0170 PROC polygon(number,length)
0180   FOR sides:=1 TO number DO
0190     forward(length)
0200     right(360/number)
0210   ENDFOR sides
0220 ENDPROC polygon
0230
```

As already mentioned, you can check your program before RUNning or LISTing it by using the SCAN instruction. (Type **scan <RETURN>** or just press **<f8>**. When you do this, COMAL will check the program structure and "learn" the procedures you have defined. If you subsequently write a defined procedure name as a direct instruction, it will be executed. This allows you to check your procedures one by one. This is a real advantage when "debugging" a program!

A few more remarks are in order: We have used the general structure described earlier with a distinct *beginning*, *middle* and *end* of the program. The *input data* is defined in lines 60 and 65, the main program is just a few lines long (80-110), and the procedures are placed at the end of the program.

In line 90 the procedure **polygon** is called. The two numbers in parentheses following the procedure name are the two variables which the procedure needs to draw the polygon. They need not have the same names as the variable names in the procedure, although they happen to in this case. It is important, however, that they are in the same instruction.

A remark is also in order about the line:

```
0190 END // of MAIN PROGRAM
```

This line is not necessary to stop the program. A COMAL program will stop when there are no more lines to execute in the main program sequence. It is included here to make the structure of the main program sequence clearer. This is largely a question of programming style. You will have strong opinions about such matters as you gain programming experience!

Saving Programs and Procedures

You may want to save your work now that we have begun to write programs which could be used again later. Please follow the instructions which apply to you:

Using a Datasette Tape Unit:

To save your program **polygon** on tape, proceed as follows:

* Place a cassette tape in your tape unit and be sure it is rewound to the beginning.

CAUTION If your tape has a *leader* with no magnetic coating on the first few inches of the tape, advance the tape for a few seconds. Otherwise you run the risk of not recording the first part of your program.

---

* Type the following direct instruction on your keyboard:
  save "cs:polygon" <RETURN>
* The message **Press record & play on Tape** will appear on your screen.
* Press **RECORD** and **PLAY** on your Tape Unit. Saving a short program like **polygon** should only take about 15 seconds.
* When your program is being saved, the screen will be blank.
* When your program has been saved, the message:
  program saved
  should appear.
* It is strongly recommended that you repeat this process, making a second *backup copy*. It will probably be most convenient to do this on the other side of your tape if you use 10 or 15 minute data cassettes. If you use longer tapes, it will probably be best to do it right after the first recording, to avoid the time-consuming rewind.

Most experienced programmers save their program file every 15 minutes or so while working. It's a good idea to save your program whenever you have completed more than you would care to lose in case of a power loss or other accident. It is wise to save 2 working copies: the *current copy* and the *previous copy*. With a tape recorder you might do this by reversing sides of your short data tape every time you save your program. That way, if something goes wrong (a power down during the save could be bad news!), you can read in the previous version to get things moving again.

When your program is completed and *de-bugged*, then you would want to make at least two copies of the final working version: an *original* working version and a *backup*.

---

* Now *label your tape*, so you know what you have! This takes a few seconds extra time now, but it could save you a hassle later, looking for a "missing program".

Using a Disk Drive

You will need to use the *storage diskette* which you prepared earlier. If you didn't do this, follow the directions for doing so in the last section of Chapter 1. Then proceed as follows:

* Insert the *storage diskette* into the disk drive.
* Now type the folowing command on your keyboard:
  save "polygon"
* The drive activity light will go on, and the drive motor will be audible for a few seconds as a copy of your program is saved to the diskette. You are free to use whatever name you wish (up to 16 characters). Of course it is wise to choose names which are descriptive and make it easy for you to find your programs again. Also, it's a good idea to include the program file name as one of the first lines of your program in a remark statement.
* To be sure that your program file has been saved as planned, type **dir** (or **cat**) and press <**RETURN**> This will show you a directory (or catalogue) of what's stored on the diskette, how many blocks each program takes up (1 block = 256 bytes), and how many blocks are unused (XXX blocks free.).
* An extra *backup* copy of all important programs should always be made on another diskette... just in case! And while you are developing a program, make a copy of the most recent version every 15 minutes or so to avoid loss of work in case of a power failure or other unexpected event! It is best to have two recent copies stored, just in case.
* Be careful to *label your diskettes* (do it at once!). That way you have a better chance of finding your programs again. Once you start writing lots of programs, your diskettes will multiply like mice!

  It is also possible to save your procedures individually. This can be done using a form of the LIST instruction. It is described in connection with the discussion of more advanced file handling in Chapter 6.

REVIEW

In this chapter you have been presented with information to help you:

* issue instructions directly from the keyboard
* correct typing errors
* use the cursor control keys
* use turtle graphics
* write simple programs using procedures
* use automatic line numbering
* use a Datassette tape unit or a disk drive for storage

You should have made a special note of the following concepts:

* 6510 (6502) microprocessor code
* high level language instructions

* direct execution vs. programmed (deferred) execution
* the total turtle trip theorem
* printing of text and numbers on the text screen
* calling of procedures
* using procedures with variables
* using a simple loop block

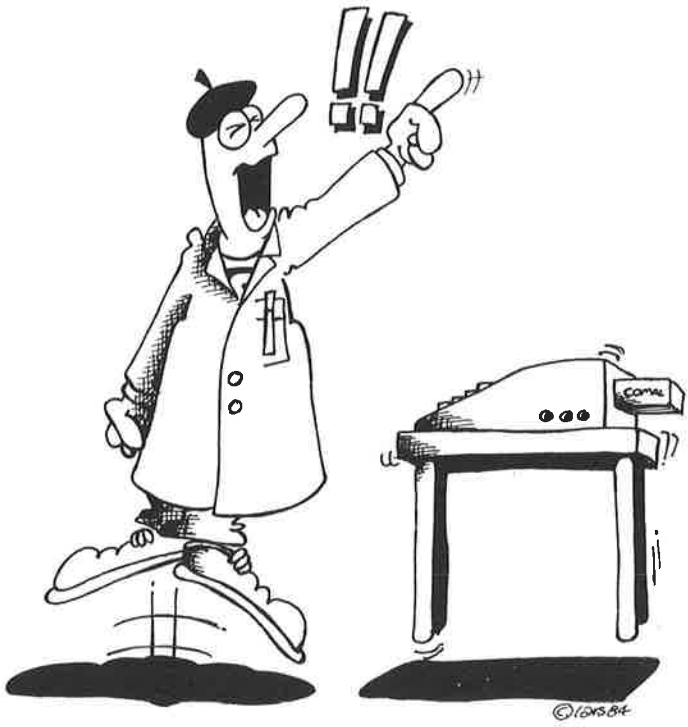
The following COMAL instructions and keywords have been presented in this chapter:

* PRINT <**text or numbers**>
* ZONE <**spacing**>
* forward(<**steps**>**)**
* back(<**steps**>**)**
* right(<**degrees**>**)**
* left(<**degrees**>**)**
* penup
* pendown
* USE <**package**>
* clearscreen
* home
* splitscreen
* showturtle
* hideturtle
* pencolor(<**color**>**)**
* background(<**color**>**)**
* setheading(<**degrees**>**)**
* WHILE - DO loops
* KEY$ **- (checks the keyboard buffer)**
* CHR$**(0)**
* AUTO - (for automatic program numbering)
* RUN - (to execute a program)
* END - (to mark the end of a program)
* // - (to insert remarks in your program)
* FOR - DO - ENDFOR loops
* INPUT "<**input prompt**>**":** <**variable list**>
* NULL - an instruction which does nothing at all!

If you have worked through this chapter, you should be prepared for the more advanced description of COMAL programming which follows in the coming chapter. It can be helpful to keep in mind that programming can really be boiled down to three fundamental elements:

* *Action blocks* are groups of instructions which input data, perform calculations, draw a picture, output data or carry out some other process in the program.

* *Loop blocks* are groups of instructions which are repeated a number of times. The FOR - DO - ENDFOR sequence and the WHILE - DO construction are two of several types of loop blocks available in COMAL.
* *Branch blocks* are instruction sequences which include decisions about which instructions to carry out next. You will learn more about this type of instruction in the next chapter.


COMAL
©Lars 84

Chapter 3

Programming with COMAL

This chapter is intended to serve as an introduction to how to use COMAL for writing programs. COMAL concepts are introduced step by step without treating each concept in depth at this stage. Examples are provided to illustrate each new concept. We will carefully comment on selected programs to explain how they operate.

We have attemped to select the examples so that they not only treat selected COMAL topics but also illustrate your Commodore 64's many facilities. Some examples have been chosen to provide a more through treatment of earlier mentioned COMAL statements. This chapter progresses from quite easy to more advanced programming techniques. The concept of the *algorithm* is introduced late in the chapter, and we have made a special effort to illustrate the power of COMAL's structured programming aids.

It is not our intention that you should be satisfied after trying our program examples and exercises. They should be considered to be guideposts to help you find your way as you begin to use COMAL. There is a great deal to be explored. Don't be afraid to strike out on your own to experiment with your own programs. You can return to the tutorial and follow it again after satisfying your curiousity. Many other books about COMAL are becoming available. Try out programs you find there or in users' group publications. More and more articles on COMAL will appear in popular computer magazines as news of this exciting language spreads. The best possible way to become proficient at this language will be to use it to write programs which can help you in your education, professional work or for entertainment.

Acquire Good Programming Habits

Everyone who writes programs will sooner or later develop his or her own programming 'style'. In the beginning, however, it can be helpful to follow a few guidelines. You may want to keep the following points in mind when you set out to solve a new programming problem:

* Type **new** to delete any earlier program from working memory.
* Then type **auto** or **auto 100** to engage automatic line numbering.
* Go right ahead with the main program. Express the problem to be solved as a list of 'procedures' to be carried out. It may be a good idea

to include them in a LOOP...ENDLOOP structure, if they are to be repeated again and again. Don't worry too much about making errors. COMAL's flexible editing facilities will make it easy to straighten things out later.

* When the structure of the main program sequence is clear, procede to begin writing the individual procedures. If a particular task is complex, break it down into smaller procedures. This technique is called 'top-down' design.
* LIST your program often to be sure that it looks like you expect it to. This will not always be the case! Use **renum** to make room for extra instructions if necessary. Don't worry about line numbers. Use **renum** often to clean things up.
* As your program nears completion, or you have completed a large procedure, execute a **scan** of your program to check for correct structure.
* After **list**ing and **scan**ning correct possible errors using the COMAL editing instructions. Check Appendix C for further information on how this is done. Be careful to make *backup-copies* of your program from time to time; this is quick and easy to do using COMAL.
* When your program appears to be error-free, try it out by typing the instruction **run**. Most often the program can be stopped again by simply pressing <RUN/STOP>. If this doesn't work, try pressing <RUN/STOP> and <RESTORE> (corresponding to "reset").
* When your program is completed and checked, save a copy on your diskette or tape for use later. The instruction **save "<programname>"** can be used if you have a disk drive, or use **save "cs:<programname>"** for a Datassette tape unit. (Don't forget to make a backup!)

Please note that in the following pages all programs are shown as they will appear after a **scan** has been issued. During program entry you need not worry about upper/lower case (except of course in text names). Nor do you need to include extra blanks to emphasize program structure. The COMAL system will take care of this for you when you scan the program.

A First Calculation

The first example illustrates how the computer handles numbers:

Program 1:

new
auto 100

```
0100 // compute an average
0110 numbera:=7
0120 numberb:=15
0130 average:=(numbera+numberb)/2
0140 PRINT "The average of the numbers"
0150 PRINT numbera;"and";numberb
0160 PRINT "is";average
0170 END
```

After entering the program check it using **scan** and **list**. Correct any errors.
Type **run** then press the <RETURN>-key (or just press <f7>).

Notes about Program 1:
The two // slashes in line 100 indicate, that the line is a comment line which the system will not process.

Computers "remember" numbers and other quantities by means of *variables*: A variable is a name which can represent a numerical value. **Program 1** contains 3 variables: **numbera, numberb** and **average.**

In line 110 the variable **numbera** is assigned the value **7**, and in line 120 the variable **numberb** is assigned the value **15.** Thus variables are given values by means of the COMAL *assignment operator* **:=**. The symbol **:=** is also called a *dynamic equals sign*.

> If you use an ordinary equality sign = when typing in a program, the COMAL system will replace it by the dynamic equals sign after a SCAN or RUN instruction has been executed.

A variable name must always begin with a letter and may consist of a maximum of 80 characters (i.e. letters, numbers or special characters). If a name is terminated with **#**, **$** or **()**, it has special meaning, as will be clarified later. The symbols **a, a#, a$** and **a()** are all considered to represent the same name within a given context.

In line 130 the *expression* **(numbera+numberb)/2** (meaning add **numbera** and **numberb**, and then divide the sum by 2) is calculated. Then this value is assigned to the variable **average.**

NB: The instruction of the variable and the expression is important. The expression on the right hand side of the assignment operator is computed first, then the variable on the left is assigned this value.

Reversing the instruction of the variable name and the expression will cause an error message to appear when the program line is entered.

Lines 140 to 160 display the result using PRINT statements. Notice how easy it is to combine numbers and text on the screen.

In line 140 the text between the quotation marks is printed.

In line 150 **the value of numbera** is printed first. Then comes the text **and**, and finally **the value of numberb**. Notice the use of the semicolon (;)

between the numbers and the text. The semicolon is not printed, but it is needed as a separation mark between the different parts of the line.

In line 160 the text **is** followed by **the value of average** is printed.

Note in connection with this example that:

* The printout starts on a new line after each PRINT statement.
* It is not the **name** of a variable but its **value** which is printed.

In line 170 the program is terminated by the statement END.

Exercises:

1. Modify the program, so that **numbera** is assigned the value **5**.
2. Try other values for **numbera** and **numberb**.
3. Add a new line to the program:
   105 PAGE
   What effect does this instruction have?
4. Place a semicolon (;) at the end of each of the lines 140 - 160. RUN the program, and note that ; yields one space between items.
5. Try to write a program which computes the average of three numbers. Be sure that the printout is correct.

The Input Statement

In the previous example we saw a program in which the computer did a numerical calculation and printed out the result on the screen. In order to compute the average of two numbers, it was necessary to change two lines in the program when each new average was to be calculated

Now we will see how to change these lines once and for all so that the program can compute the average of any two numbers we choose without changing the program every time.

Program 2:

Program 2 is available on the demo diskette. You can copy it into working memory by using the instruction **load "Program 2"**, or type it in as follows:

```
new
auto 100

0100 // computing an average
0110 INPUT "Enter the 1. number ": numbera
0120 INPUT "Enter the 2. number ": numberb
0130 average:=(numbera+numberb)/2
0140 PRINT "The average of the numbers"
0150 PRINT numbera;"and";numberb
0160 PRINT "is";average
0170 END "end!"
```

Check that the program is correct, then execute it using the command RUN.

List the program and notice how using the INPUT statement allows the program variables to be assigned a value while the program is being run.

Thus it is not only possible to print out variable values from a program, but also to read values into a program.

Notes:

* Program execution is stopped by an INPUT statement until the user responds. In **Program 2** it is necessary to type in a number in response to each INPUT statement followed by a <RETURN>.
* The text of the INPUT statement must be terminated by a colon (:) before the variable. All other characters will result in error messages.

Exercises:

1. Add a line with the instruction PAGE to the program, so the screen is cleared at the beginning of a run.

2. It is also possible to send the output to a printer, if available.
   Add the lines
   135 SELECT OUTPUT "lp:"
   165 SELECT OUTPUT "ds:"
   Run the program again and see what happens.
   Line 135 directs the output to the printer, and line 165 brings output back to the display screen.

3. Write a program which computes the average of 3 numbers. The numbers should be read in using INPUT statements.

Circles

The output from a program can also be in the form of a drawing. The next program draws circles.

Program 3:

```
new
auto 100
0100 // circles are drawn
0110 PAGE
0120 INPUT "Enter the 1. radius ": radiusa
0130 INPUT "Enter the 2. radius ": radiusb
0140 sumradius:=radiusa+radiusb
0150
0160 USE graphics
0170 graphicscreen(1)
0180 circle(160,100,radiusa)
0190 circle(160,100,radiusb)
0200 circle(160,100,sumradius)
0210
0220 WHILE KEY$=CHR$(0) DO NULL
0230 END
```

Check the program to be sure it is correct, then run it.

The program consists of an input section and a calculation section which is separated from the printout section by the **empty line** 150. Empty lines can be useful for separating various parts of a program to make the program structure clearer.

Lines 160 and 170 are necessary to prepare the computer for doing graphics.

Lines 180-200 draw 3 circles all of which have their centers at screen coordinates (160,100), i.e. about in the middle of the screen.

The radii of the three circles are obtained in lines 120-140. If the radius exceeds 99 units, the circle will overlap the edge of the screen.

The statement in line 220 is described in Chapter 2. Its purpose is to keep the graphics screen visible until the user presses any key.

The function **KEY$** is useful for reading in characters from the keyboard while a program is running. We will treat this function again later.

Note:

It may turn out that the "circles" look more like egg-shaped curves than circles. This phenomenon is due to the adjustment of the screen displays height/width ratio. If an adjustment is available, you may wish to make use of it so that circles appear correctly on the screen.

Exercises:

1. Correct the program so that the third circle is drawn with a radius equal to the difference between the two radii. You should also change the name of the variable **sumradius**!
2. Experiment with the use of other arithmetic operations in line 140.
3. Move the centers of the circles.
4. Add instructions so that more circles with other radii and centra are drawn.

5. The center of the circles can also be read in as an input statement.
   For example add the line:
   135 INPUT "Center: X,Y = ": xc,yc
   Correct lines 180-200 to:
   180 circle(xc,yc,radiusa)
   190 circle(xc,yc,radiusb)
   200 circle(xc,yc,sumradius)
   Run the program.
   Note that it is necessary to respond with two values separated by a comma (,) in the new INPUT statement.
6. The circles can be filled with colors. Use the instruction **fill(x,y)** to do this, where **(x,y)** must be the coordinates of a point inside the closed figure which is to be colored in.
   For example if **Program 3** is extended with the lines:
   202 pencolor(2)
   204 fill(160,100)
   the innermost circle will be colored red. Try it!
7. Try to color other regions of screen by changing the coordinates in line 204.
   For example change line 204 to:
   204 fill(0,0)
   What happens?
8. Now try to color other areas on the screen. Change the number in the **pencolor** instruction in line 202 to employ other colors. See the color code table in Appendix B.

Procedures I

When writing extensive COMAL programs, it is particularly important to make use of *procedures*:

A procedure is a "subprogram" which can be called from the main program or from another procedure. It can perhaps best be illustrated by means of some examples. **Program 4** is available on the demo diskette (and tape), or it may be typed in:

Program 4:

```
new
auto 100

0100 //filled circles and squares
0110 start'graphics
0120 draw'square(10,10,300,180,brown)
0130 draw'circle(160,100,70,yellow)
0140 draw'square(100,50,50,50,purple)
0150 draw'circle(125,75,20,orange)
0160
0170 WHILE KEY$="" DO NULL
```

```
0180 END
0190
0200
0210 PROC start'graphics
0220   USE graphics
0230   graphicscreen(1)
0240   brown:=8
0250   yellow:=7
0260  purple:=4
0270   orange:=10
0280 ENDPROC start'graphics
0290
0300 PROC draw'square(xmin,ymin,xside,yside,color)
0310   pencolor(color)
0320   moveto(xmin,ymin)
0330   draw(xside,0)
0340   draw(0,yside)
0350   draw(-xside,0)
0360   draw(0,-yside)
0370   xpoint:=xmin+.5*xside
0380   ypoint:=ymin+.5*yside
0390   paint(xpoint,ypoint)
0400 ENDPROC draw'square
0410
0420 PROC draw'circle(xcenter,ycenter,radius,color)
0430   pencolor(color)
0440   circle(xcenter,ycenter,radius)
0460 ENDPROC draw'circle
```

Run the program; afterwards we'll take a look at how the program works.

Program 4 consists of:

The main program	(lines 100-180)
Three procedures:	
start'graphics	(lines 210-280)
draw'square	(lines 300-400)
draw'circle	(lines 420-460)

Notice that a procedure is called by its name, sometimes followed by parentheses with a list of parameters to be transferred to the procedure.

The procedure itself is built up as follows:

```
PROC <name>(<a>,<b>,<c>,...)
<statement 1>
<statement 2>
...
...
...
ENDPROC <name>
```

Recall that sharp brackets < > around a word mean that the word and the brackets can be replaced by names or statements of the users choice: E.g. <**name**> could be replaced by the name **start'graphics**, **printout** or something else describing the purpose of the procedure. The notation <**statement no**> stands for a valid COMAL statement.

The main program consists of a comment line followed by 5 lines which all call procedures.

In line 110 the main program just calls the procedure with the name **start'graphics**, and the computer proceeds to execute the statement in this procedure.

When the computer has carried out the statements in the procedure, it returns to the main program and goes on to the next line.

In line 120 the procedure with the name **draw'square** is called. In this case it is not only called by name but also with a pair of parentheses containing some numbers. The numbers are separated by commas (,).

There must be **exactly** just as many numbers in the call as there are variables in the parentheses following the procedure name.

```
draw's    quare(10  ,10  ,300  ,180  ,brown)
PROC      draw'square(xmin,ymin,xside,yside,color)
```

Notes:

- The variable **brown** has the value 8. It received that assignment during the execution of the procedure **start'graphics**.
- During the execution of **draw'square** the procedure will use these values:
  - **xmin:=10**
  - **ymin:=10**
  - **xside:=300**
  - **yside:=180**
  - **color:=brown (:=8)**
- Now the computer can carry out the instructions in the procedure **draw'square**, for the values of all variables are now available.
- The procedures **draw'square** and **draw'circle** consist of a sequence of graphics instructions. Use the index to find detailed descriptions of these instructions.
- Next the procedure **draw'square** computes the midpoint of the square in lines 370 and 380.
- When the computer has completed execution of the procedure **draw'square**, it returns to the next line in the main program.
- In line 130 the procedure **draw'circle** is called then executed.
- In lines 140 and 150 the procedures are called again, but this time other parameter values are used.
- A procedure can be called many times with various parameter values if desired. This is one of the great advantages of using a procedure.

Exercises:

1. Try to move the circles and squares around the screen by changing the two first numbers in the procedure calls. These numbers stand, respectively, for the center of the circle and the lower left corner coordinates of the square.

   For example try moving the last square and circle into the middle of the screen:

   ```
   140 draw'square(135,75,50,50,purple)
   150 draw'circle(160,100,20,brown)
   ```
2. The lengths of the sides of the squares can also be changed. Change the circles' radii.
3. Add other colors. See the color codes in Appendix B.
4. Other circles and squares can be drawn by adding new program lines to the main program containing procedure calls. Try it.
5. Try writing a procedure yourself which can draw a triangle and fill it up with a color. Add a program line which calls your procedure.

COMAL and text

The next example, **Program 5**, is also composed of a main program which calls two procedures:

Main program	(100 - 160)
Procedure **read'in**	(190 - 260)
Procedure **print'out**	(280 - 460)

Before we enter and try out this program, we must be familiar with the concept of a *string*.

A *string constant* is a text enclosed in quotation marks. E.g. **"John"**, **"billing code"** and **"he has 7 seals"**.

So far all the variables we have worked with have been number variables. It is also possible to define variables which contain sequences of letters, special characters and digits. Such variables are called *string variables*.

String variables can always be recognized because they end with a dollar sign ($). Examples of string names are:

name$, city$, country$

When a string is to be assigned a value, a *declaration statement* must occur early in the program to assure that enough room is reserved in memory for the string. This is also refered to as *dimensioning* the string variable.

Examples:

DIM name$ OF 20	(room for up to 20 characters)
DIM city$ OF 25	(room for up to 25 characters)
DIM country$ OF 40	(room for up to 40 characters)

Now the string variables may be assigned text values (string contants):

```
name$:="Jonathan Doe"
city$:="London"
country$:="England"
```

Notes:

* Text must always be enclosed between quotation marks (").
* The text need not be as long as the maximum space specified in the declaration statement.
* A text variable can contain both large and small letters, spaces, digits and certain special characters **(,./<>?!#$%'+-:;=)**. On the Commodore 64 it can also include the graphics symbols. When we refer to *characters* we mean any of the above.

  In **Program 5** we will practice the use of procedures and learn more about strings and string variables. In addition we will also try using the semigraphics characters of the computer. They can be seen on the front side of most keys. See Appendix D for more about the use of the keyboard.

  Pay particular attention to the procedure **print'out** if you will be typing in the program instead of reading it from the demo diskette or tape:
* Line 310: 2 spaces and 36 <**C**= o> characters.
* Line 320: 2 spaces, 1 <**C**= j>, 34 spaces and 1 <**C**= l> character.
* Line 400: 2 spaces and 36 <**C**= u> characters.

  (NB: <**C**= o> means: hold down the Commodore key, while pressing the o-key.)

Program 5:

```
new
auto 100

0100 // read'in and print'out of text
0110 DIM name$ OF 25
0120 DIM from$ OF 25
0130 DIM text$ OF 30
0140 read'in
0150 print'out
0160 END
0170
0180
0190 PROC read'in
0200   PAGE
```

```
0210    PRINT "Write a message:"
0220    INPUT "The letter is to ": name$
0230    INPUT "The letter is from ": from$
0240    PRINT "The message can fill one line."
0250    INPUT "Start here:": text$
0260 ENDPROC read'in
0270
0280 PROC print'out
0290    PAGE
0300    PRINT
0310    PRINT " ---------------------------------"
0320    PRINT " !                                !"
0330    PRINT " !                                !"
0340    PRINT " !                                !"
0350    PRINT " !                                !"
0360    PRINT " !                                !"
0370    PRINT " !                                !"
0380    PRINT " !                                !"
0390    PRINT " !                                !"
0400    PRINT " ---------------------------------"
0410    PRINT AT 4,6: "To ";name$
0420    PRINT AT 6,6: text$
0430    PRINT AT 8,6: "Best regards"
0440    PRINT AT 9,6: from$
0450    CURSOR 20,1
0460 ENDPROC print'out
```

In the main program the first statements declare the variables **name$**, **from$** and **text$**. Then the procedure **read'in** is called. It allows for the input of values for the text variables.

When the read-in procedure is completed, the computer returns to the main program. In the next line execution is directed to the procedure **print'out**, which prints out the message inside a frame.

Notes:

* A new version of the **PRINT** statement is used:
  PRINT AT <line>,<column>.
  E.g. in line 440, where the **from$** text is specified to begin on line 9, in column 6. This syntax makes it possible to place text or numbers anywhere on the screen.
* Line 450: **CURSOR 20,1**
  CURSOR <line>,<column> places the cursor anywhere on the screen, but no message is printed.
* See also **INPUT AT**, which is used in **Program 10**.

Exercises:

1. Run the program a few times with different messages to get an idea of how the program operates.
2. If a printer is available, one can get a hard copy of the text screen by pressing <CTRL P>:

When the program has finished running, and the text is ready on the screen, press **P** while holding down the <**CTRL**>-key.
3. Try revising the program so that text variables can be read in and printed at various positions on the screen.

Here is a BRIEF REVIEW of the foregoing information on strings and text:

1. A computer can work with numbers or with words. This is done using number variables and text variables. Text variables can be recognized because they always end with $.
2. Variables can be given values:
   * by assignment statements :=
   * in **parentheses** in procedure calls
3. Text can be written on the screen by means of PRINT statements. (It can also be done in other ways, e.g. in the text segment of an INPUT statement, as we have seen.)
4. Drawings can be made on the screen using graphics instructions from the graphics packages (**use graphics** or **use turtle**), or by means of the semigraphics character set, which is shown on the front of the keys.
5. If a program is more than a few lines long, it should be composed using *procedures*. A procedure is a 'sub-program' which can be used many times from the main program or from other procedures. We'll be studying more on the use of procedures later in this chapter.

Branching. Conditional Execution

The computer can also distinguish between expressions, which are **true** or **false**. Such expressions are called *logical expressions*. Some examples:

7=2 is a logical expression, which both we and the computer would consider **false**.

23<54 is a **true** logical expression.

Whether or not the logical expression **number>10** is **true or false** can not be determined before we know the value **number**.

COMAL contains the two *logical constants* TRUE and FALSE, which have numerical values **1** and **0** respectively.

In the following examples we have illustrated how the computer can be made to execute various statements according to whether a logical expression is true or false.

Program 6:

```
new
auto 100

0100 // find the maximum
0110 PAGE
0120 PRINT "The maximum of two numbers:"
0130 PRINT
0140 INPUT "Write the 1st number ": a
0150 INPUT "Write the 2nd number ": b
0160
0170 maximum:=a
0180 IF maximum<b THEN maximum:=b
0190
0200 PRINT
0210 PRINT "Maximum is ";maximum
0220 END
```

The new construction occurs in line 180: **IF - THEN**

It is an example of a *branch*, also called *conditional execution*. In this case the construction means:

"IF the variable **maximum** is less than the variable **b**, THEN **maximum** is set equal to **b**".

The computer evaluates the logical expression **maximum<b**.

IF it is true, the computer will execute the statement following the instruction THEN. This is often described by saying: *the condition* between IF and THEN must be fulfilled.

If the condition is not fulfilled, the computer simply proceeds on to the next program line.

It is often the case, however, that it is desirable to have several statements executed when the condition is fulfilled, while other statements should be executed if it isn't. This situation is handled in COMAL by using a new structure:

IF - THEN - ELSE - ENDIF.

IF <condition> **THEN**
<statement 1>
<statement 2>
...
...
ELSE
<statement a>
<statement b>
...
...
ENDIF

Lines 170 - 180 in **Program 6** could thus also be written as follows using this IF-construction:

```
170 IF a<b THEN
172    maximum:=b
174 ELSE
176    maximum:=a
180 ENDIF
```

Program 7:

```
new
auto 100

0100 // right or wrong
0110 DIM text$ OF 10
0120 PAGE
0130 PRINT "Guess my number: 1, 2 or 3"
0140 INPUT "Try your luck ":answer
0150
0160 RANDOMIZE
0170 my'number:=RND(1,3)
0180
0190 IF answer=my'number THEN
0200    text$:="CORRECT"
0210 ELSE
0220    text$:="WRONG"
0230 ENDIF
0240
0250 PRINT
0260 PRINT "My number was ";my'number
0270 PRINT "The guess was ";answer
0280 PRINT
0290 PRINT "So the guess was ";text$
0300 END
```

Notes on this program:

* Lines 190-230: Note the IF - THEN - ELSE - ENDIF structure, described earlier.
* Lines 160-170: the computer is able to generate a random number with the instructions RANDOMIZE and RND:

  RANDOMIZE causes the computer to position a pointer at a "random" position in an array of random numbers. (The present COMAL version executes an automatic RANDOMIZE even if the statement RANDOMIZE is left out.)

  In **my'number:=RND(1,3)** the variable **my'number** is set equal to a random (RaNDom) value 1, 2 or 3.

  The range of numbers can be changed. E.g. **RND(-10,10)** will randomly generate one of the numbers: -10,-9,-8,...,0,...,8,9,10.

Exercises:
1. Experiment using other number ranges in the RND function.
2. Try changing the statement RANDOMIZE to RANDOMIZE 1 and run the program several times. What happens?

The CASE Structure

If one must distinguish among many conditions at the same time, then the **CASE structure** is advantageous to use. It is built up as follows:

```
CASE <variable> OF
WHEN  <1st value>
     <statement 1a>
     <statement 1b>
     ...
     ...
WHEN  <2nd value>
     <statement 2a>
     <statement 2b>
     ...
     ...
(additional WHEN-values)
     ...
     ...
OTHERWISE
     <statement a>
     <statement b>
     ...
     ...
ENDCASE
```

If e.g. <variable> equals <2nd value>, then execution proceeds in the corresponding segment of instructions: <statement 2a> - <statement 2b>, etc. Then execution continues in the line after ENDCASE.

If <variable> does not equal any of the given WHEN values, then execution continues with the statements in the OTHERWISE segment. OTHERWISE and the statements in the corresponding segment are optional.

This structure is used in the following example, where one can choose among several different exercises in computation.

Each exercise is given in a procedure. An answer to an exercise is evaluated in the procedure **result**, which is therefore called from each exercise-procedure:

Main program	**exercise1 - result**
	exercise2 - result
	exercise3 - result
	exercise4 - result

Program 8:

```
new
auto 100


0100 // Computation exercises
0110 PAGE
0120 PRINT "Choose an exercise:"
0130 PRINT
0140 INPUT "Which number (1 - 4) ": number
0150
0160 CASE number OF
0170 WHEN 1
0180   exercise1
0190 WHEN 2
0200   exercise2
0210 WHEN 3
0220   exercise3
0230 WHEN 4
0240   exercise4
0250 OTHERWISE
0260   PRINT "You have chosen an incorrect number."
0270 ENDCASE
0280
0290 END
0300
0310
0320 PROC exercise1
0330   PRINT
0340   INPUT "INT(7.3+3.2 DIV 2) = ": answer
0350   correct:=INT(7.3+3.2 DIV 2)
0360   result(correct,answer)
0370 ENDPROC exercise1
0380
0390 PROC exercise2
0400   PRINT
0410   INPUT "3-30/2+12 = ": answer
0420   correct:=3-30/2+12
0430   result(correct,answer)
0440 ENDPROC exercise2
0450
0460 PROC exercise3
0470   PRINT
0480   INPUT "4.25+2.5/5*2 = ": answer
0490   correct:=4.25+2.5/5*2
0500   result(correct,answer)
0510 ENDPROC exercise3
0520
0530 PROC exercise4
0540   PRINT
0550   INPUT "34 MOD 10-2*5 = ": answer
0560   correct:=34 MOD 10-2*5
0570   result(correct,answer)
0580 ENDPROC exercise4
0590
```

```
0600 PROC result(correct,answer)
0610   PRINT
0620   PRINT "The answer is: ";answer
0630   PRINT "The correct answer is: ";correct
0640   PRINT
0650   IF answer=correct THEN
0660     PRINT "Your answer is right!"
0670   ELSE
0680     PRINT "Wrong.  Please try again..."
0690     PRINT "Check Appendix C: calculating with COMAL."
0700   ENDIF
0710
0720 ENDPROC result
```

Notes:
A procedure may be called from another procedure, as well as from the main program. For example **result** is called from the **exercise** procedures.

Exercises:
1. Try responding to some of the exercises in the program.
2. Create a new exercise 5:
   Write a procedure **exercise5**.
   Add the new WHEN value in the CASE structure.
   Remember to change the INPUT statement.
3. Write a program which prints out different messages. The messages should depend on the value of the variable which is entered.

Repetition and Loops

Repetition is one of the fundamental building blocks of programming. The computer is uniquely well-suited for repeating operations over and over again. In COMAL there are several different statements which can accomplish repetition. These statement combinations are classified as *loop blocks* or simply as *loops*.

The first example shows how the computer be made to repeat a set of instructions a certain number of times:

Do <these statements> 100 times.

This is accomplished with a FOR - ENDFOR loop:

```
FOR <no>:=<start> TO <end> DO
    <statement a>
    <statement b>
    ...
    ...
ENDFOR <no>
```

Statements a, b and so on are repeated (**<end>-<start>+1**) times, if <start> and <end> are integers:

the first time <no> equals <start>
the second time <no> equals <start>+1
the third time <no> equals <start>+2

.

.

.

the last time <no> equals <end>

Program 9:

```
new
auto 100

0100 // Investigation of RND
0110 USE graphics
0120 graphicscreen(0)
0130 wrap
0140 window(0,1000,-10,10)
0150 moveto(1000,0); drawto(0,0)
0160
0170 FOR no:=0 TO 1000 DO
0180   number:=RND(-10,10)
0190   moveto(no,0); draw(0,number)
0200 ENDFOR no
0210
0220 WHILE KEY$=CHR$(0) DO NULL
0230 END
```

The program illustrates graphically how "random" numbers generated by the RND function can be distributed. Notice the loop block:

line 170-200: the FOR - ENDFOR statement.
The loop is executed 1001 times.

The statement can be extended using the STEP parameter:

FOR <no>:=<start> TO <end> STEP <steps> DO

where STEP causes <no> to take on the values: <start>, <start+ steps>, <start+2*steps> etc.
The loop ends when <no> passes <end>.

If the STEP parameter is left out (as we have done so far), then STEP is automatically set equal to 1.

In addition to the graphics statements which we already have become acquainted with, the program contains some new statements. Their use is explained in detail in Chapter 5 in the section on graphics.

Finally we can take a closer look at the statement in line 220. This is another example of repetition:

In the WHILE - DO statement, the computer checks the keyboard again and again, until any key is activated.

The keyword KEY$ is a function which outputs the last character which was sent from the keyboard. If no key has been pressed, then "" (ASCII code 0) is returned. KEY$ will thus continue to return "" until any key is pressed.

```
while <no key is pressed> do <nothing>
WHILE   KEY$=CHR$(0)   DO   NULL
```

But the most common use of the WHILE statement is in a loop block extending over several lines:

```
WHILE <condition> DO
    <statement a>
    <statement b>
    ...
    ...
ENDWHILE
```

If the <condition> between WHILE and DO is fulfilled, the computer goes ahead with statements a, b, etc. These statements are executed one after the other. If something occurs in the statements so that the condition is no longer fulfilled, then program execution jumps from the WHILE-DO line to the line just after ENDWHILE, next time the WHILE-DO line is considered.

See the word WHILE in the index to find a more detailed description of how this construction can be used.

Another often encountered (perhaps the most simpel) loop structure is the REPEAT - UNTIL construction:

```
REPEAT
    <statement a>
    <statement b>
    ...
    ...
UNTIL <condition>
```

The statement list is repeated until the <condition> is fulfilled.

In the next example, **Program 10**, this type of loop determines how long the user can continue to guess the letters in a "secret" word. The example also illustrates the use of strings in COMAL.

The program structure:
The main program - **select'word**
- **new'letter**

Program 10:

```
new
auto 100

0100 // word guessing
0110 PAGE
0120 select'word
0130 number:=0
0140
0150 REPEAT
0160    number:=number+1
0170    new'letter
0180 UNTIL answer$=remember$
0190
0200 PRINT AT 20,5: "Now finished"
0210 PRINT AT 21,5: number;"letters have been used."
0220 END
0230
0240
0250 PROC select'word
0260    DIM name$ OF 20, letter$ OF 1
0270    DIM used$ OF 200
0280    INPUT "New word: ": name$
0290    length:=LEN(name$)
0300    DIM answer$ OF length, remember$ OF length
0310    answer$:="----------------------"
0330    used$:=""
0340    PAGE
0350    PRINT "GUESS THIS";length;"LETTER WORD"
0360    PRINT AT 8,5: "Word: ";answer$
0370 ENDPROC select'word
0380
0390 PROC new'letter
0400    INPUT AT 10,5,1: "New letter ": letter$
0410    used$:=used$+letter$
0420
0430    position:=letter$ IN name$
0440    IF position>0 AND position<=length THEN
0450       answer$(position):=letter$
0460       name$(position):="#"
0470    ENDIF
0480
0490    PRINT AT 10,17: " "
0500    PRINT AT 8,5: "word: ";answer$
0510    PRINT AT 12,1: used$
0520 ENDPROC new'letter
```

Lines 150-180: the REPEAT - UNTIL loop:
When the user has the answer which the computer remembers, the program continues in line 190.

Notes:

* Line 160: the variable **number** occurs on both sides of the assignment

operator :=. This is legal (and often done). Remember how the assignment operator works: First the expression on the right hand side of the sign is computed. Then the variable on the left side is assigned the value computed.

* Line 400: **INPUT AT 10,5,1** means that the INPUT statement must begin on line **10**, column **5**, and there must be room for **1** character in the the answer field. Try to write several answers to see how the program works. Try changing **1** to **3**, and run the altered program.
* The branch construction IF - ENDIF begins in line 440 and extends over several lines, ending in line 470.
* Line 440: **AND** is an example of a *logical operator*. It requires that both conditions in the IF - THEN statement must be fulfilled.

Note particularly about strings:

* Line 290: The **LEN** function indicates how many characters are included in the word. This is how the **length** of the word is determined.
* Line 300: It is possible to use variables in **DIM** statements.
* Line 410: Words can be 'added together' using the + character. This process is called *concatenation* of strings.
  Example: **"cat"+"fish"** yields the word **"catfish"**.
* Line 430: **IN** is a logical operator which acts on strings. It indicates the first position of the first character in the search string.

  Examples:
  "ok" IN "cooking" yields the value **3**.
  "i" IN "cooking" gives the value **5**.

  If the search string is not contained in the given text string, then the value will equal **0** (zero).

  Examples:
  "salt" IN "cooking" gives the value **0**.
  "sing" IN "cooking" gives the value **0**.

* Line 450-460: One can select particular substrings in a text by using the position of the substring in the text.

  Example:
  LET **text$:="cooking"**
  text$(3) is the letter **"o"**.
  text$(4:7) is the string **"king"**.

* In line 460 the letter found is replaced by a character which never will occur in a word. This is done to allow the same letter to occur more than once in a word. In this case the character selected is #.

Exercises

1. The program is quickly simplified to fill in the guessed letter at all its positions in the secret word.

   Make the changes:
   430 FOR position:=1 TO length DO
   440 IF letter$=name$(position:position) THEN
   475 ENDFOR position

   Try your new program.

2. If you have changed the program according to exercise 1 the variable **remember$** and linie 460 are superfluous. Can you make the program work with these corrections?
3. Where are you to make changes to be able to play with secret words of more than 20 letters? Try.

Arrays. Indexed Variables

When you have to work with lots of numbers, it can become time consuming to read them all in and give them different names. Sometimes at least 100 variable names may be needed when solving one of the following problems, for example:

* Computing the average of 100 numbers
* Determining the maximum and minimum of 100 numbers
* Sorting 100 different numbers

Large collections of numbers can be handled in COMAL by declaring an *array* using a dimension statement as for example the following:

DIM x(50)

This statement reserves room for 50 numbers in the computer's memory. Each variable will have the same name **x** but a different number:

x(1), x(2), x(3),...., x(49), x(50)

Such variables are also termed *indexed variables* with the number of each variable called an *index*.

It is possible (but not common practice) to give each of the indexed variables a value using an assignment statement:

```
x(1):=23
x(2):=71
x(3):=-12.45
   .
   .
x(49):=6
x(50):=0.852
```

In the next program example we will work with indexed variables which are assigned values by means of an INPUT statement.

The program draws line segments through the coordinates of a number of points.

Program 11 consists of:

a read-in section	**(lines 110-220)**
a graphics section	**(lines 270-300)**

Program 11:

```
new
auto 100

0100 // line segments
0110 DIM x(50), y(50)
0120 PAGE
0130 PRINT "A line is drawn through the points."
0140 PRINT
0150 REPEAT
0160   INPUT "Number of points: ": number
0170 UNTIL number>=2 AND number<=50
0180 PRINT
0190 FOR no:=1 TO number DO
0200   PRINT "Enter x(",no,"),y(",no,"):";
0210   INPUT "": x(no),y(no)
0220 ENDFOR no
0230 PRINT
0240 PRINT "Press any key to draw the figure."
0250 WHILE KEY$=CHR$(0) DO NULL
0260
0270 USE graphics
0280 graphicscreen(0)
0290 moveto(x(1),y(1))
0300 FOR no:=2 TO number DO drawto(x(no),y(no))
0310 WHILE KEY$=CHR$(0) DO NULL
0320 END
```

Notes:

* Line 110: Room is reserved for 50 pairs of x- and y-coordinates.
* Line 160: The program inquires in an INPUT statement how many sets of coordinates to be read in. The INPUT statement is included in a REPEAT - UNTIL loop which also ensures that at least 2 sets of

coordinates are entered. (A line can't be drawn if only one set has been entered.)

* Lines 190-220: the coordinate pairs **x(1),y(1) x(2),y(2)... x(number),y-(number)** are entered in a FOR - ENDFOR loop.
* In line 270-300 the figure is drawn using graphics statements.

Exercises:

1. Use the program with a few points.
2. Add a line in the program which will place a small circle around each point. For example try **circle(x(no),y(no),3)**.
3. Write a program which computes the average of an arbitrary number of values. The program should include the following sections:
   Enter the number of values. Enter the values in the array of numbers. Compute the sum of the numbers. Average := the sum/number of values.
4. Those arrays which we have handled so far have been arrays with one index. They are termed *one-dimensional* arrays.
   In COMAL an array can have two or more dimensions. For example:
   DIM bookcase (3,4)
   The variable **bookcase** is a two dimensional array. One can imagine a bookcase with 3 shelves, each with room for 4 items.:

56	**17**	**-3**	**72**
89	**0.5**	**14**	**94**
8	**-6**	**78**	**66**

   For example with the above values for the elements of the array:
   bookcase(2,3)=14 and **bookcase(3,1)**=8

Try changing **Program 11** so that the one-dimensional arrays **x()** and **y()** are replaced by a two-dimensional array **point(,)**. You can begin by changing line 110 to **DIM point(50,2)**.

Make changes in lines 290-300 yourself.

Text Arrays

We are not restricted to the declaration of arrays of numbers. We can also declare arrays which contain strings:

DIM message$(8) OF 20

Room is made of 8 **message$**'s, each up to 20 characters in length:

message$(1):="Remember the sun."

.

.

message$(8):="Hurrah! Hurrah!"

Just as number arrays, text arrays can have two or more dimensions. The next program illustrates the use of a 2-dimensional text array. The array is declared in line 130:

DIM person$(50,4) OF 30

It is to be used as an address list for up to 50 persons, with 4 items of information about each one:

person$(no,1):="<name>"
person$(no,2):="<street>"
person$(no,3):="<town>"
person$(no,4):="<telephone number>"

In this program we will also become acquainted with yet another way to read in variable values: a DATA statement.

Information can be stored in DATA statements which can be read using READ statements.

The following statements:

READ number,item$,x,points
DATA 17,"doll",-346,10

replace four separate assignment statements:

number:=17
item$:="doll"
x:=-346
points:=10

Notice here that numbers and strings can be mixed in the same DATA and READ statements.

The following program consists of

Lines 120-250:	dimensioning and assignments
Lines 270-350:	printout of information which agrees with the search code
Lines 380-500:	DATA statement

Program 12:

```
new
auto 100

0100 // address list
0110 PAGE
0120 number:=50; no:=0
0130 DIM person$(number,4) OF 30, text$ OF 30
```

```
0140 DIM found(number)
0150 REPEAT
0160   no:+1
0170   FOR information:=1 TO 4 DO READ person$(no,information)
0180 UNTIL EOD
0190 number:=no
0200
0210 INPUT "Search for: ": text$
0220 FOR no:=1 TO number DO
0230   information:=0
0240   REPEAT
0250     information:=1
0260     found(no):=text$ IN person$(no,information)
0270   UNTIL found(no)>0 OR information=4
0280 ENDFOR no
0290
0300 PRINT
0310 PRINT "Persons whom the search key fits:"
0320 PRINT
0330 FOR no:=1 TO number DO
0340   IF found(no)>0 THEN
0350     FOR information:=1 TO 4 DO PRINT person$(no,information)
0360       PRINT
0370     ENDIF
0380 ENDFOR no
0390 END
0400
0410 DATA "Susan Hansen","Lindebakken 13"
0420 DATA "Silkeborg","06-841723"
0430 DATA "Commodore Data","Bjerrevej 67"
0440 DATA "8700 Horsens","05-641155"
0450 DATA "Jan Mogensen","Skovgade 4"
0460 DATA "1717 Copenhagen","01-456701"
0470 DATA "Knud Jensen","Sneglevej 12 D"
0480 DATA "2820 Gentofte","secret"
0490 DATA "Wesleyan University","Physics Department"
0500 DATA "Middletown CT 06457","(203) 344-7930"
```

Notes:

* The READ statements need not be placed together with the DATA statements. The first READ instruction in the program begins by reading in the first value in the first DATA statement no matter where it occurs in the program. (This can be altered. See the discussion in Chapter 4 on READ and DATA.)
* In line 180 the function EOD is used to terminate the reading process. The value of EOD is **0** (i.e. **false**), until the last data value is read in. Then COMAL sets it equal to **1** (i.e. **true**). When the UNTIL condition thus is fulfilled, the program continues in line 190.

Exercises:

1. Try out the program. Try to understand how it operates. Try respon-

ding to **Search for:** with just <RETURN>. Add new DATA statements.

2. Replace the values in the DATA statements with others of your own choosing. The program can of course also be used to file any information you may choose. For example you might exchange the variable **person$** with a new variable **item$** which could represent items in an inventory. For example:
   ```
   item$(no,1):="warehouse"
   item$(no,2):="storage area"
   item$(no,3):="shelf"
   item$(no,4):="item"
   ```
3. Add a line to the program which prints out the classification number of the person or item along with the other information.
4. Add further information about each person in the address list:
   ```
   DIM person$(number,5)
   ```
   where for example:
   ```
   person$(no,5):="<profession>"
   ```

Procedures II

In the section *PROCEDURES I* we became acquainted with two different ways of using procedures:

WITHOUT passing of parameters

```
//main program
<statements>
...
name
<statements>
...
END
//
PROC name
<statements>
...
ENDPROC name
```

WITH passing of parameters

```
//main program
<statements>      ...
name(4,"Christina")
<statements>
...
END
//
PROC name(number,text$)
<statements>
...
ENDPROC name
```

If there is a transfer of parameters in parentheses, then the number and type must be in agreement:

```
name     (4  ,"John",from,x() ,logo$)
PROC     name(number,text$,start,no(),string$)
```

The number and type of the *actual* parameters in the procedure call must correspond to the number and type of the *formal* parameters in the procedure's parentheses.

4,"John",from,x(),logo$ are the actual parameters.
number,text$,start,no(),string$ are the formal parameters.

If the parameters are in agreement with respect to number and type, they need not have the same name.

We have emphasized that procedures should be used when building up programs, because:

* Procedures can be used again and again in different parts of the program.
* The program will be clearer to read, more logical and easier to grasp if it has been broken down into procedures with well-chosen names.
* Procedures can be saved in a procedure library on disk or cassette tape for use later in other programs.

There are many ways to use procedures. In the following sections you will find an introduction to the extended use of procedures and functions:

* In what ways are they similar?
* In what ways are they different?
* How can they be used.

Local and Global Names

In COMAL one must distinguish between *global* and *local* names. A local variable name - in contrast to a global name - is only defined and recognized in a limited segment of the program. For example:

```
FOR no:=2 TO number DO
    <statements>
    ...
ENDFOR no
```

The variable name **no** is local in the FOR - ENDFOR loop. It is undefined outside this loop.

In connection with procedures one also refers to local names, only recognized within the procedure, and global names which are recognized throughout the program. In general, parameters listed in parentheses after a procedure name are local. In addition the procedure may contain other global and local parameters.

The advantage of local names is that they do not interfere with other parts of the program and vice versa.

Enter, run and examine the next example with global and local variable names. Note the values of the quantities which are printed out.

Program 13:

```
new
auto 100

0100 // local variables
0110 a:=1;b:=1
0120 PRINT a;b
0130 local'global(4)
0140 PRINT a;b
0150 END
0160
0170 PROC local'global(a)
0180    PRINT a;b
0190 ENDPROC local'global
```

In the parameter transfers examined so far we have seen a number of one-way transfers **from** the main program **to** a procedure. In order to permit transfer of local parameters **from** the procedure, the parameters must be declared using a REF prefix. The procedure in the following example shows how this can be done.

Program 14:

```
new
auto 100

0100 PROC minmax(a,b,REF min,REF max)
0110    // minimum and maximum are found
0120    IF a<b THEN
0130       min:=a; max:=b
0140    ELSE
0170       min:=b; max:=a
0180    ENDIF
0190 ENDPROC minmax
```

A main program which uses this procedure might look like this:

```
0010 //main program
0020 t:=23
0030 s:=-41
0040 minmax(t-s,t+s,minimum,maximum)
0050 PAGE
0060 PRINT "t-s =";t-s;"and";t+s =";t+s
0070 PRINT "Minimum, maximum:";minimum;maximum
0080 END
```

Exercises:

1. The names are unimportant. Exchange the variable names **minimum** and **maximum** with **a** and **b** respectively. Note that they have no effect on the results. (A change like this is easiest to make using the command CHANGE: **change "minimum","a"**, etc.)
2. After a procedure has been typed in and checked using the SCAN command, it can be used as a direct instruction.
   Type the following directly from the keyboard:

   ```
   scan
   minmax(12/7,7/12,x,y)
   print x;y
   ```

   Try using other values, and try using other procedures as direct instructions.

3. Make the following changes and run the program:

   ```
   100 PROC minmax(REF a,REF b)
   185 a:=min;b:=max
   ```

   and

   ```
   40  minmax(t,s)
   70  is deleted
   ```

   Note, that the variables **t** and **s** change their value in the procedure.
   Now the procedure can no longer be used in the form **minmax-(67,78)** with constants in the call. But it can be used in the form **minmax(x,y)** if the variables **x** and **y** have been given values in advance:

   ```
   scan
   x=1236;y=251
   this=(x+y)/x;that=(x-y)/y
   minmax(this,that)
   print "Minimum, maximum: ";this;that
   ```

   Experiment with the legal as well as the illegal version.

A particularly elegant property of procedures is that they can call one another. A procedure can even call itself. Such a procedure is called a **recursive** procedure.

The next program shows an example of such a procedure using graphics.

Program 15:

```
new
auto 100

0100 // concentric filled circles
0110 USE graphics
0120 graphicscreen(1)
0130
0140 draw'circle(160,100,100,2)
0150
0160 WHILE KEY$=CHR$(0) DO NULL
0170 END
0180
0190 PROC draw'circle(xc,yc,r,color)
0200   pencolor(color)
0210   circle(xc,yc,r)
0220   paint(xc,yc)
0230
0240   IF r>10 THEN draw'circle(xc,yc,r-10,color+1)
0250
0260 ENDPROC draw'circle
```

In line 240 the procedure **draw'circle** calls itself until **r** gets too small.

Functions

COMAL's built-in standard functions can be used in computations. We have already used standard functions like PI, RND, INT, LEN. See Chapter 4 for information on other standard functions.

Just as it is possible to define procedures using the construction:

PROC - ENDPROC

you can define your own functions in COMAL using the structure:

FUNC - ENDFUNC

Procedures and functions have many properties and uses in common. The next program shows how functions can be defined and used to find the roots of analytical functions. The program also employs some standard functions.

Overview:

main program	(lines 100-350)
function **round**	(lines 380-400)
function **f**	(lines 420-440)

where the functions are built up using the following structure:

```
FUNC <name>(<number>)
    <statement a>
    <statement b>
    ...
    ...
RETURN <computed'expression>
    ...
ENDFUNC <name>
```

An understanding of the theory behind the method to be used requires some knowledge of mathematics. However this is not essential in order to use the program or to understand the statements which compose it.

Within the discipline of "informatics" the word *algorithm* is sometimes used to describe a formula or a means of computation. It is an important part of good programming practice to provide a complete description of the algorithm on which a program is based. The description can be given in greater or lesser detail depending upon who will use the program. A minimum requirement is of course that the programmer must be able to understand it later on, if the program must be corrected or revised.

There are in fact many tragic examples of substantial waste of resources, both in government and in private industry, due to poor documentation of programs.

Program description:

1. The program searches for roots using the *midpoint method*.
2. The program is designed to find a solution to the equation **f(x)=0**, where **f** is a function which is continuous in the region of interest.
3. The user must be able to provide an initial guess of two numbers **a** and **b** with the property that **f(a)** and **f(b)** have opposite signs. If this condition is not fulfilled, the program will request other numbers.
4. The midpoint between **a** and **b** is found, and the value of the function in this point is determined.
5. If the value of the function is sufficiently close to zero, then the program will conclude that the midpoint is a root. This approximation to the root will be printed, and the program will stop.
6. Otherwise the program will continue comparing the signs of values of the function:

If the value of the function in the midpoint has the same sign as the value of the function in **a**, then the root which is sought is assumed to lie between the midpoint and **b**. Therefore the midpoint is set equal to the new **a** value as the search proceeds.

If on the other hand the value of the function in **a** and the value of the function in the midpoint have opposite signs, then there must be a root between **a** and the midpoint. The midpoint therefore becomes the new **b** endpoint.

7. The program then returns to step 4.
8. In this fashion the interval around the root is narrowed down until the root has been found within the required uncertainty, or the program is interrupted by pressing <STOP>.

Program 16:

```
new
auto 100
0100 // solving the equation f(x)=0
0110 PAGE
0120 error:=1e-04
0130 REPEAT
0140  INPUT ”End point values A,B: ”: a,b
0150 UNTIL SGN(f(a))=-SGN(f(b))
0160
0170 LOOP
0180  sign'a:=SGN(f(a))
0190  sign'b:=SGN(f(b))
0200  xmid:=(a+b)/2
0210  ymid:=f(xmid)
0220  IF ABS(ymid)<error THEN
0230   PRINT
0240   PRINT ”A solution to the equation =”;round(xmid)
0250   STOP
0260  ELSE
0270   PRINT ”.”;
0280   IF SGN(ymid)=sign'a THEN
0290    a:=xmid
0300   ELSE
0310    b:=xmid
0320   ENDIF
0330  ENDIF
0340 ENDLOOP
0350 END
0360
0370
0380 FUNC round(number)
0390  RETURN INT(number*10000+.5)/10000
0400 ENDFUNC round
0410
0420 FUNC f(x)
0430  RETURN 3*x*x+2*x-5
0440 ENDFUNC f
```

The function **f(x)** itself is defined in the structure **FUNC f(x)**. It is defined here by means of the expression **3*x*x+2*x-5**. Thus the program must find solutions to the equation:

3*x*x+2*x-5 = 0

Experiment with other functions besides this one when trying out the program.

Notes:

* A new COMAL loop structure: **LOOP - ENDLOOP** (continuous repetition) is introduced.
* Clarity is enhanced by the use of descriptive names.
* The standard function **SGN(<expression>)**:
  SGN(<expression>)=1 , if <expression> is greater than 0
  SGN(<expression>)=0 , if <expression> equals 0
  SGN(<expression> <expression> is less than 0
* The standard function **ABS(<expression>)** returns the numerical value of the expression. E.g. **ABS(-2)** equals **2**.
* The function **round** rounds off the expression for **number** to 4 decimal places.
  E.g. **round(3.141593)** equals **3.1416**
  If 3 decimal places are required, then 10000 can be replaced by 1000 in this procedure, etc.
  There should be a correspondence between the required accuracy of the calculation specified by the variable **error** and the rounding accuracy specified in the function **round** by choosing e.g. **10000.** At the very least no more decimal places than those represented by the value of **error** should be returned.

Exercises:

1. Run the program with various functions **f(x)**.
   Test the program first using functions with well known roots e.g. 2x-6.
   Use the program to solve equations which can not be solved by means of ordinary analytical methods:
   The equation **EXP(x)**=x+7 is an example of such a problem. It is called a "transcendental" equation. It can be solved using this program by defining the function **f** to be **EXP(x)-x-7**.

2. Functions can also be used as direct instructions when they have been SCANed. Try for example:
   scan
   print round(2.71828183)
3. Create a numerical function **FUNC average(a,b)**, which returns the average of **a** and **b**. Try it as a direct command.

4. Write a function **FUNC vowels(text$)**, which counts the number of vowels in a given string. Try using it as a direct instruction. Hint: take a look at **Program 17** for inspiration.

String Functions

Functions can be used for other purposes than just calculating matematical expressions (a job which they of course do very well).

The functions which we have just worked with are numerical functions. COMAL can also handle *string functions*. A string function is a function which outputs a string instead of a number. Just as the case of string variables, the name of string functions must end with the character $.

KEY$** is an example of a built in standard string function which is already available in COMAL. Others include **STR$(327) which changes the numerical constant **327** to the string constant **"327"**.

The following program illustrates how you can create your own string functions. It consistes of a brief main program and the function **separate$**. This string function is designed to separate a string into vowels and consonants.

Program 17:

```
new
auto 100

0100 PRINT separate$("COMAL string functions")
0110 END
0120
0130
0140 FUNC separate$(a$)
0150  // consonants or vowels
0160  long:=LEN(a$)
0170  FOR i:=1 TO long DO
0180   IF a$(i) IN "aeiouAEIOU" THEN
0190    a$:=a$(i)+a$(1:i-1)+a$(i+1:long)
0200   ENDIF
0210  ENDFOR i
0220  RETURN a$
0230 ENDFUNC separate$
```

Try this example:
If **a$:**="testing" and **i:=2;** then line 190 will act as follows:

a$= "e" + "t" + "string"

Notes:

* The vowels are placed in reverse order.
* COMAL can interpret an expression such as **a$(7:6)**. This is used in line 190 when **i:**=long. (But note that **a$(8:6)** is undefined.)

* **a$(i+1:)** means: from the (i+1)'th letter to end of word
* **a$(:i-1)** means: from beginning to the (i-1)'th letter

Exercises:

1. Try out the program to see that it works as it should. Choose other strings to test the program. You might want to experiment with special cases like "a", "iiiiiiieeeee", "qwrtp" and the empty string.
2. Create a string function which reverses the order of the letters in an arbitrary string. Try it out!
3. After having been SCANed a string function can be used as a direct command just as a numerical function. For example:
   scan
   print separate$("sodapop and icecream")
4. Create a string function **FUNC fillup$(number,letter$)** which prints **number** of the same **letter$**. Try it out as a direct command: **print fillup$(30,"*")**.

Closed Procedures

If you want to be completely certain that any name conflicts between variable names in procedures and the main program will be avoided, then you can CLOSE your procedures or functions. When you do so, you make all variable names in the procedure or function local. Only those values which are given in parentheses after the name of the procedure are allowed in or out.

This is accomplished by using the instruction **CLOSED**. For example:

PROC name(number,text$) CLOSED

It can be very useful to be able to close a procedure. This is particularly true when you want to save a very general procedure in a procedure library and use it in many different situations. It can be difficult to remember the names of all the variables which were used. By closing the procedure you can get around this problem.

The next program illustrates a general procedure which can be used to sort any series of numbers. The numbers will be sorted so that they are ordered by increasing value. For example **4, 3, 7, -1** are sorted to **-1, 3, 4, 7**.

The sorting method is called the *bubble sort*.

There are many algorithms available for sorting. For example on the demonstration diskette and on the tape you will find the program **quick-.sort**. It is a fast and efficient sorting program.

The bubble sort used in **Program 18** in not the most efficient method, but it is interesting and easy to understand:

Consider the numbers in pairs starting at the beginning of the sequence. If a larger number precedes a smaller one, then they will be swapped. Now the next pair (the second and third) is considered. These two numbers are swapped, if the largest number comes first and so on down the sequence. The procedure is repeated until no more swaps occur. Here is a brief illustration of the process:

1st run-through:

4 3	7	-1	is changed to	**3 4**	7	-1
3	**4 7**	-1	no change			
3	4	**7 -1**	is changed to	3	4	**-1 7**

2nd run-through:

3 4	-1	7	no change			
3	**4 -1**	7	is changed to	3	**-1 4**	7
3	-1	**4 7**	no change			

3rd run-through:

3 -1	4	7	is changed to	**-1 3**	**4**	**7**
-1	**3 4**	7	no change			
-1	3	**4 7**	no change			

On the next run-through there will be no more exchanges.

main program	(lines 100-290)
procedure **print'out**	(lines 320-370)
procedure **swap**	(lines 390-420)
procedure **bubble'sort**	(lines 440-610)

All the procedures are closed.

Program 18:

```
new
auto 100
0100 DATA 2,4,78,45,23,-2,56,45,199,43
0110 DATA 3,0,100,34,-19,34,67,88,4,10
0120
0130 // data read-in
0140 DIM position(100)
0150 no#:=0
0160 REPEAT
0170   no#:+1
0180   READ position(no#)
0190 UNTIL EOD
0200
0210 PAGE
0220 PRINT "Unsorted number:"
0230 print'out(no#,position())
0240 PRINT
0250 PRINT
0260 bubble'sort(no#,position())
```

```
0270 PRINT "Sorted number :"
0280 print'out(no#,position())
0290 END
0300
0310
0320 PROC print'out(total,number()) CLOSED
0330  // total number in sequence number() is printed out
0340  ZONE 8
0350  FOR no#:=1 TO total DO PRINT number(no#),
0360  ZONE 0
0370 ENDPROC print'out
0380
0390 PROC swap(REF a,REF b) CLOSED
0400  // a and b are swapped
0410  remember:=a; a:=b; b:=remember
0420 ENDPROC swap
0430
0440 PROC bubble'sort(total,REF number()) CLOSED
0450  // number() is sorted in increasing numerical order
0460  IMPORT swap
0470
0480  REPEAT
0490   no'swap:=TRUE
0500   FOR no#:=1 TO total-1 DO
0510    IF number(no#)>number(no#+1) THEN
0520     no'swap:=FALSE
0530     swap(number(no#),number(no#+1))
0540    ENDIF
0550   ENDFOR no#
0560  UNTIL no'swap
0570 ENDPROC bubble'sort
```

In line 460 of **bubble'sort** the statement **IMPORT** is used. It can be used to make variables or procedures accessible in an otherwise closed procedure. In this case the procedure name **swap** is made available in the procedure **bubble'sort**.

In the main program in line 340 the instruction **ZONE 8** is used to space the printout in columns. Printout of a row of numbers separated by a comma (,) in PRINT-statements will be done in columns 8 spaces wide.

Note:

* The DATA statements are placed near the beginning of the main program. They are easy to find when changing to new values.

Exercises:

1. Try out the program with the values provided. Then try with your own values. You should also try the program with special cases like **DATA 2** or **DATA 3,3,3,3,3,3**.
2. This exercise deals with *external procedures:* If a disk drive is available, procedures can be saved on diskette individually. Later on

they can be brought in to be used in other programs when needed. After use they are removed from a program.

Such procedures are termed *external* when they are available outside program memory, as on a diskette.

There are two conditions which external procedures must fulfill:

a. They must be CLOSED

b. They must not contain IMPORT statements.

First save the program as backup. Now remove all other program lines from **program 18** except for the procedure **print'out** and save **print'out** on diskette as a **prg** file under the file name **ext.print'out**:

save "ext.print'out"

The prefix **ext.** has been added to distinguish this type of file from other information in the disk directory.

Then delete the procedure **print'out** from **program 18**, and add a line with a declaration which indicates that the program will use an external procedure:

300 PROC print'out(no,position()) EXTERNAL "ext.print'out"

Now run the program, and note that the external procedure is fetched from the diskette twice during program execution.

The use of external procedures saves room in memory. On the other hand the disk operations take time, so the method should only be used for larger programs or for programs in which the delay time is not important.

3. Write a program which sorts words in alphabetical order.

Only a few corrections of **Program 18** are necessary to accomplish this task:

First change the following lines:

140 DIM t$(100) OF 20

510 IF t$(no#)>t$(no#+1) THEN

Next use the CHANGE instruction:

CHANGE "position","t$"

CHANGE "number","t$"

Supply all the variables in the procedure **swap** with $ signs, and change the contents of the DATA statements to words or other text.

COMAL can still interpret the logical expression in line 510, because a 'word' consists of a sequence of characters each of which has an ASCII value. See Appendix A for a list of ASCII codes.

The computer handles the letters in each word one after the other when two words are compared. If the first letters of both words are the same, then the next pair is compared, and so on. This allows an evaluation of which word is 'largest'. For example the word **"apple"** is 'less than' **"banana"**, because **a** comes before **b** in the alphabet, and **banana** is less than **baseball**, because **n** comes before **s**.

Be careful when comparing words containing both upper and lower case letters. Try some experiments!

File Handling

We have seen how it is possible to save a copy of a program on diskette or on a cassette tape using the command SAVE. A copy of the saved program can be fetched into the working memory later using the instruction LOAD.

There are also other means of saving programs and program segments. See Chapter 4 under the heading LIST - ENTER - MERGE for more information about this. In Chapter 6 you will find a summary of these file operations.

The next program illustrates one of the many ways in which data can be saved. By 'data' we mean lists of numbers or text or perhaps a mixture of numbers and text. Data can be stored in a *file*. A more complete treatment of the use of files in COMAL including numerous examples is found in Chapter 6.

The introductory program which we will consider here consists of:

The main program
the procedures
file'numbers(<fileno>,<filename$>,number(),total)
fetch'numbers(<fileno>,<filename$>,REF number())

The two procedures take care of the jobs of saving numerical data on disk or cassette and retrieving the data again.

The main program is simply a test program which saves some numbers in a file, fetches them again and prints them on the screen.

These procedures operate by opening a *data stream* to or from a region on the diskette. The data stream is characterized by the number <fileno>, and the region on the diskette is characterized by its <filename$>. It is thereafter possible to 'write' to the data stream, if it has been opened in the WRITE mode, or one can 'read' from the data stream, if it has been opened in the READ mode. A data stream remains 'open' until it is 'closed'.

Saving data:

OPEN FILE <fileno>,<filename$>,**WRITE**

PRINT FILE <fileno>: number

CLOSE FILE <fileno>

Fetching data:

OPEN FILE <fileno>,<filename$>,**READ**

INPUT FILE <fileno>: number

CLOSE FILE <fileno>

Program 19:

```
new
auto 100

0100 PROC file'numbers(fileno,filename$,number(),total)
0110  OPEN FILE fileno,filename$,WRITE
0120  FOR i:=1 TO total DO
0130   PRINT FILE fileno: number(i)
0140  ENDFOR i
0150  CLOSE FILE fileno
0160 ENDPROC file'numbers
0170
0180 PROC fetch'numbers(fileno,filename$,REF number())
0190  OPEN FILE fileno,filename$,READ
0200  i:=0
0210  REPEAT
0220   i:+1
0230   INPUT FILE fileno: number(i)
0240   PRINT number(i);
0250  UNTIL EOF(fileno)
0260  CLOSE FILE fileno
0270 ENDPROC fetch'numbers
0280
0290
0300 // numbers are saved and read in from a file
0310 DIM number(100)
0320 PAGE
0330 PRINT "Enter numbers, each followed by <RETURN>."
0340 PRINT "Terminate by entering 99999:"
0350 no:=0
0360 REPEAT
0370  no:+1
0380  INPUT "": number(no);
0390 UNTIL number(no)=99999
0400 no:-1 // the last number is not saved
0410 PAGE
```

```
0420 FOR I:=1 TO no DO PRINT number(i);
0430 PRINT
0440 PRINT "PRESS ANY KEY TO WRITE TO THE FILE"
0450 WHILE KEY$=CHR$(0) DO NULL
0460
0470 file'numbers(2,"@0:numberdata",number( ), no)
0480
0490 PAGE
0500 PRINT "PRESS ANY KEY TO FETCH DATA AGAIN"
0510 WHILE KEY$=CHR$(0) DO NULL
0520 PAGE
0530
0540 fetch'numbers(3,"@0:numberdata",number())
0550
0560 END
```

Notes:

* Note that you are free to place procedures first in the program
* If the data are to be saved to a cassette tape, the file name must be supplemented with the **cs:** unit indicator:
  "cs:numberdata"
* Data must be fetched using the same file name as the one under which they were saved. The stream number need not be the same.

The advantage of saving data in files is that the data need not be associated with a particular program as with DATA statements. The same data can be used by many different programs.

Notice especially about **file'numbers**: In the procedure call it is essential to specify the (**total**) number of data elements which are to be saved.

But note regarding the procedure **fetch'numbers** that the computer will simply stop reading in numbers from the file when no data are left. To register this condition the function **EOF(<fileno>)** is very useful. It takes on the value **TRUE** when the file contains no more data, thereby fulfilling the UNTIL condition.

Data can be saved in ASCII-code format by means of the **PRINT FILE** instruction. The **INPUT FILE** instruction must then be used to enter the data. This combination can be used both with a disk drive and with a Datassette unit. If you are using a disk drive it will usually be best to use the **WRITE FILE** and the **READ FILE** instructions instead, because data can be saved more quickly and more compactly in binary form than in ASCII form.

Exercises:

1. Try out the program with arbitrary numbers. Change the file names and stream numbers. Check for legal stream numbers.
2. Use the program to create a set of data. Use these numbers instead of the numbers in the DATA statements in **Program 18**. You will have to

delete lines 100-200 and replace them by lines which read in the numbers from one of the data files which we have just worked with.
3. Write a program which saves strings in a file. Read the information from the DATA statments in **Program 12** into this file. Then use this file instead of the DATA statements in **Program 12**.

Error Handling

It is important that programs are constructed so that they do not 'crash', if the user does something unexpected.

One of the most common causes of undesired program interruption is the entry of LETTERS in an INPUT statement in which a NUMBER is expected.

In COMAL there is an error handling structure which can take this problem and many others into account. Note that the use of this structure is treated more completely in the reference section, Chapter 4. Here we will concentrate on the one type of error mentioned above.

The structure is:

```
TRAP
.
(statements in which errors are expected)
.
HANDLER
.
(statements to be executed in case of an error)
.
ENDTRAP
```

If an error occurs in the statements between TRAP and HANDLER, the computer will jump to the statements between HANDLER and ENDTRAP. At the same time an ERR code will be generated. The ERR code can be used to determine which of the statements in the HANDLER-section should be executed.

In the next program example an error handling structure has been placed in the LOOP - ENDLOOP loop which we used earlier. This loop assures that the INPUT-statement will be executed again if input errors are detected.

Note the following about the various COMAL loop-structures:

* In the WHILE - ENDWHILE structure the condition is placed right after WHILE at the beginning of the loop.
* In the REPEAT - UNTIL loop the condition is placed at the end, right after UNTIL.
* In the LOOP - ENDLOOP structure a condition can be placed anywhere inside the loop using the EXIT WHEN command. When the

condition is fulfilled, execution passes to the first statement after ENDLOOP.

The LOOP - ENDLOOP structure:

LOOP

.

.

EXIT WHEN <condition> (or just **EXIT** with no condition)

.

.

ENDLOOP

The program consists of a general read-in procedure with error handling and a brief main program used to check out the procedure.

Program 20:

```
new
auto 100

0100 PROC number'input(line,pos,dpos,text$,REF number)
0110  // number-safe input
0120  // only <STOP> interrupts program
0130
0140  LOOP
0150
0160   TRAP
0170
0180    PRINT AT line,pos: SPC$(LEN(text$)+dpos);" *"
0190    INPUT AT line,pos,dpos: text$: number
0200
0210    EXIT // if the input is OK
0220
0230   HANDLER
0240
0250    CASE ERR OF
0260    WHEN 2
0270     PRINT AT 24,1: "The number was too big."
0280    WHEN 206
0290     PRINT AT 24,1: "A number is expected."
0300    OTHERWISE
0310     PRINT AT 24,1: "What happened?"
0320    ENDCASE
0330    FOR pause:=1 TO 1000 DO NULL
0340    PRINT AT 24,1: SPC$(25)
0350
0360   ENDTRAP
0370
0380  ENDLOOP
0390
0400 ENDPROC number'input
0410
0420
```

```
0430 // test of input errors
0440 PAGE
0450 REPEAT
0460  number'input(10,3,10,"Type in a number: ",number)
0470  PRINT AT 12,3: SPC$(15)
0480  PRINT AT 12,3: number
0490 UNTIL FALSE
0500 END
```

Notes:

* The statement **EXIT** instructions the computer to jump out of the LOOP structure if the input is ok.
* The string function **SPC$(number of spaces)** can be useful for clearing part of the screen.
* Line 180 clears the INPUT field and places a * two blank spaces after the end of the field.

Exercises:

1. Try out the program using both numbers and letters. Try pressing **<RETURN> with no input.**
2. The LOOP structure can be replaced by a REPEAT loop. The following lines can be used:

   no'error:=FALSE
   REPEAT
   no'error:=TRUE
   UNTIL no'error

   Where should these lines be inserted?
3. Replace the CASE error texts the the system error message ERRTEXT$: **PRINT AT 24,1: ERRTEXT$**.
4. Your final examination:
   The character * in line 180 is a special detail. What can happen, if this character is left out? Experiment!

   After working through this tutorial chapter you should be well prepared to continue developing your skill with the COMAL programming language. Of course there is still much more to be learned, and you can run into situations which have not been covered here.

   In Chapter 4 you will find a complete reference section treating all of the many commands and statements in COMAL. In Chapter 4 you will find explanations of each instruction with examples to illustrate its use.

Chapter 4

COMAL Overview

Commands used before and during Program Entry

NEW - AUTO - RENUM

NEW

is a command which causes the program and the data in working memory to be deleted. System variables are set to their initial values, and packages and associated variables are also deleted.

AUTO

is a command which sets up automatic line numbering during program entry. The range of legal line numbers is: 1 - 9999. During program entry each line should be terminated by pressing <RETURN>. The system will automatically print the next line number on the screen. AUTO can be disengaged by pressing <RUN/STOP>. If AUTO is engaged again (or engaged after manual entry of part of a program), automatic line numbering will begin with the last line number in the program + 10.

Examples:

AUTO	Gives line numbering: 10, 20, 30,...
AUTO 1000	Gives line numbers: 1010, 1020,...
AUTO 100,2	Gives line numbers: 100, 102, 104,...

Notes:
Line numbering with intervals of 10 is often appropriate, for it allows the insertion of several extra lines between existing line numbers.

If a line number already exists, the number will appear in reversed characters to warn the user against unwanted overwriting of existing code.

RENUM

is a command which provides the program in working memory with new line numbers. Renumbering can begin from any line in the program.

Examples:

RENUM	New numbering: 10, 20, 30,...
RENUM 2000,5	New numbering: 2000, 2005, 2010,...
RENUM 300;4000,10	Line numbers from and including 300 will be changed to: 4000, 4010,...

Commands Which are Used for Program Editing

EDIT - FIND - CHANGE - DEL - SCAN

EDIT

is a command which causes program lines to be printed one at a time without indentation. It is particularly useful for correction of program lines which take up more than one line on the screen. If the LIST instruction is used, some lines may contain unwanted spaces after the end of the first line. After editing, pressing <**RETURN**> will cause the next program line to appear, if more than one line edit has been requested.

Examples:

EDIT	allows editing of all lines, one at a time.
EDIT 130	allows line 130 to be edited.
EDIT 210-290	permits editing of lines 210 - 290.
EDIT colorcodes	lets the user edit the procedure **colorcodes.**

Note:
The EDIT command can only be used for printout to the screen or to a printer.

FIND

is a command used during editing to find a name or text segment in a program. When the text segment has been found, the system prints out the program line with the cursor placed on the first character of the text. After possible corrections press <RETURN>, and the system will search for the next occurence of the text.

Examples:

FIND "John"	The system will search the entire program for the word **John**.
FIND 200-500 "John"	The system searches for the word **John** in lines 200 - 500.
FIND colorcodes "red"	The system searches for the word **red** in the procedure **colorcodes**.

CHANGE

is a command which is used to search for and replace a text segment. When the text segment to be changed has been found, the system prints out the program line with the text segment blinking like a cursor.

There are now three options:

1. You can make the change by pressing <**RETURN**>.
2. You can edit the line without the automatic change:
   Press the <**C=**> key.
   Change the line as desired.
   Press <**RETURN**>.
   The search will be continued.
3. You can order the search to continue with no changes:
   Press **n** or **N**.
   The search will continue.

Press <**STOP**> to interrupt the CHANGE operation.

Examples:

CHANGE "red","yellow"	The search text **red** is replaced by the replacement text **yellow** everywhere in the program.
CHANGE 50-200 "x1","xstart"	The change is made in lines 50 - 200.
CHANGE square "up","right"	The change is made in the procedure **square**.

DEL

is a command which is used to delete program lines.

Examples:

DEL 20	Line 20 is deleted.
DEL 40,200-280	Lines 40 and 200 - 280 are deleted.
DEL printout	The procedure **printout** is deleted.

SCAN

is a command which causes the system to run through the program in the working memory. This process is also called making a *prepass*. The program structure is checked for possible errors, and any error in structure is reported. After a SCAN without any error messages, approved procedures and functions can be executed directly from the keyboard like commands.

Examples:
Program as entered:

```
0100 number=0
0110 repeat
0120 print number
0130 number:+2
0140 print "You saw some even numbers."
0150 end
```

SCAN
The system will report: **at 150: "UNTIL" missing**

add the line: **135 until number>20**

After a new SCAN the program should appear as follows:

```
0100 number:=0
0110 REPEAT
0120   PRINT number
0130   number:+2
0135 UNTIL number>20
0140 PRINT "You saw some even numbers."
0150 END
```

Other Commands

SETEXEC

is a command which has two distinct formats: **SETEXEC-** and **SETEXEC+**.

During the initiation of the system, a **SETEXEC-** is executed. This causes the keyword EXEC to be omitted from the listing of procedure calls.

After a **SETEXEC+** command EXEC will be printed before all procedure calls.

Example:
Program segment as it would be listed after system start-up:

```
0100 PRINT "Numbers are read in and printed out."
0110 read'in
0120 print'out
0130 END
0140
0150 PROC read'in
0160   INPUT "Write the number: ": number
0170 ENDPROC read'in
0180 PROC print'out
0190   PRINT number
0200 ENDPROC print'out
```

After **SETEXEC+** the lines 110 -120 are changed to:

```
0110 EXEC read'in
0120 EXEC print'out
```

Commands Used to Check Available Memory and Disk Storage

SIZE - CAT - DIR

SIZE
is a command which causes the present usage of bytes of working memory to be reported.

Example:
SIZE
System response:

prog	**data**	**free**
13501	02466	14747

CAT
is a command which causes a catalogue of the contents of the diskette to be printed. If several disk drives are connected, then the device number can be included in the command.

Examples:

CAT	All file names are listed.
CAT "t*"	The names of all files beginning with **t** are listed.
CAT "?est??"	The names of all files which are 6 characters long and with characters 2-4 equal to **est** are listed.
CAT "*=seq"	All names of sequential files are listed.
CAT "2:"	The contents of the diskette in the second disk drive are listed. The second drive must be set up as "device 9". This can be done using a jumper inside the second drive or by means of software. See your 1541 instruction manual for more on how to do this.

Note:
Pressing the space bar will stop the printout of the disk catalogue. Pressing it again will allow it to continue. <STOP> will end it.

DIR

may be used as a command or as a statement. Like CAT this instruction causes the contents of the diskette in the drive selected to be printed out. Unlike CAT, DIR can be used as a statement in a program if desired.

LIST - ENTER - MERGE - DISPLAY

LIST

is a command which is used to print out all or part of the program in working memory. It is also used to store all or part of a program to diskette or to the Datassette tape unit. When this is done, the program is saved as a sequential file in ASCII-format. Copies of the program which have been saved using the LIST command must be reentered using the ENTER or MERGE commands. They can NOT be entered using LOAD.

Examples:

LIST	All program lines are printed.
LIST 200-400	Program lines 200-400 are printed.
LIST 300-	The program is printed from line 300 onward.
LIST demoproc	The procedure with the name **demoproc** is printed.

If the LIST instruction is followed by a name in quotation marks, then the listing will be done to diskette or cassette tape:

LIST "program name"	The entire program is saved under the file name **program name**.
LIST demoproc "lst.demo"	The procedure **demoproc** is saved under the file name **lst.demo**. The prefix **lst.** is not essential. It is included to remind us that the program has been saved by a LIST command.

Notes:

The printout of the listing to the screen will proceed more slowly if the <CTRL> key is depressed during the printout.

The printout can be stopped temporarily by pressing the space bar once. Press it again to continue the listing.

Pressing the <**STOP**> key interrupts the printout.

The printout can be directed to a printer, if available, with the command **list "lp:"**

If a program line extends beyond a single line on the screen, the LIST instruction will cause it to be split due to indentation. Place the cursor on the line in question and press <CTRL-A>. The line will be pulled together again with no indentation.

ENTER

is a command which fetches a program which has previously been saved to diskette or cassette tape using the LIST command into working memory. **Note:** ENTER acts differently than MERGE. If there is already a program in working memory, ENTER will erase it.

Examples:

ENTER "lst.name"	The program **lst.name** is fetched from diskette.
ENTER "cs:lst.Program 3"	The program **lst.Program 3** is fetched from the Datassette unit.

Note:
A program which has been saved using the SAVE instruction can NOT be read in again using ENTER.

MERGE

is a command which is used to fetch a program segment from diskette or cassette and copy it into working memory. The program segment must have been stored using the LIST command.

Examples:

MERGE "lst.circumference"	The program **lst.circum ference** is fetched from diskette and added to the existing program with line numbers starting after the end of the current program.
MERGE 1000,5 "lst.start"	The program (or segment) **lst.start** is read in and added to the current program at lines 1000, 1005, 1010...

Be careful not to unintentionally overwrite existing program lines.

DISPLAY

is a command which lists a program or a program segment with NO LINE NUMBERS in the listing.

Examples:

DISPLAY	The entire program is listed to the screen.
DISPLAY 20-90 "lp:"	The program from line 20 to and including line 90 is printed on the lineprinter with no line numbers.
DISPLAY sort "dsp.sort"	The contents of the procedure **sort** is stored on diskette under the name **dsp.-sort**.

Note:
A program which has been saved on diskette (or tape) with the DISPLAY command can not be fetched again using ENTER or MERGE. However it can be read in as an ordinary sequential ASCII file using the instruction INPUT FILE.

SAVE - LOAD

SAVE

is a command which saves a copy of the program in working memory to diskette or tape in compact binary form. A SAVEd program can be fetched later using one of the following: LOAD, RUN or CHAIN.

Examples:

SAVE "program name"	The program in working memory is saved to disk under the file name **program name**.
SAVE "cs:racetrack"	The program is saved to cassette tape under the file name **racetrack**.

Note:
Any program packages which are associated with the COMAL program by means of the LINK instruction are saved together with the COMAL program as one file. When the program is later entered into working memory, e.g. using LOAD, both the COMAL program and the machine language package are read in together.

LOAD

is a command which transfers a copy of a program from diskette or cassette tape into working memory. The program must have been saved earlier by means of the SAVE command. The LOAD command deletes any previously existing program and all variables from working memory.

Examples:

LOAD "program name"	transfers a copy of the program saved under the file name **program name** from diskette into working memory.
LOAD "cs:"	A copy of next program on the tape is fetched into memory via the Datassette.

RUN - CHAIN - CON

RUN

is a command which causes the program in working memory to be executed. All variables are zeroed and the computer begins by examining the program structure for possible errors. A program can also be fetched from diskette or tape and started automatically using the RUN command.

Examples:

RUN	Program execution is started (the program is 'run').
RUN "program name"	The file **program name** is fetched from diskette and execution begins.

CHAIN

can be used as a statement or as a command. It fetches a copy of a program from diskette or from cassette tape and starts it running. Any existing program in working memory will be deleted first.

Used as a command CHAIN "<file name>" works like RUN "<file name>".

CHAIN is particularly useful when used as a statement in a program. It allows the user to break down a large program into smaller independent units.

Examples:

CHAIN "cs:name"	The program **name** is fetched from cassette tape and started.

Program example:

```
INPUT "Choose a program number: ":no
CASE no OF
WHEN 1
  CHAIN "program 1"
WHEN 2
  CHAIN "program 2"
OTHERWISE
  CHAIN "program 3"
ENDCASE
```

CON

is a command which causes program execution to continue in an interrupted program. The program may have been interrupted by an error, by activation of the STOP key or by a STOP statement in the program. While the program is stopped, changing the contents of existing variables is permitted. However new variable names may not be added, and the program may not be changed. No line may be altered, and no new lines may be added to the program while it is interrupted. If this is done, execution cannot be continued using the CON command.

STATUS - STATUS$

STATUS is a command which causes the system to report on the status of the disk operating system and zero the *error flag*. STATUS$ is a string function which contains the status report. STATUS performs the same operation as PRINT STATUS$.

Example:
Right after the disk system is turned on
STATUS
will cause the system to answer
73,cbm dos v2.6 1541,00,00
depending on disk drive used.

VERIFY

is a command which can be used to check that the program on the diskette or cassette tape (saved using the SAVE command) is identical to the program which is currently in the working memory of the computer.

Warning: Take care not to change the program in working memory before using VERIFY (spell correctly!).

Example:

VERIFY "test prog" — The COMAL system reports **verify error**, if the program saved under the file name **test prog** and the program in working memory are not exactly alike.

COPY - DELETE - RENAME - PASS

COPY

can be used as a command or a statement for copying diskette files.

Examples:

COPY "old'file","new'file" — The system makes a copy of the program **old'file** and saves it on the same disk drive under the name **new'file**.

COPY "0:program 3","1:program 3" The system copies **program 3** from disk drive **0:** and saves it with the same name on disk drive **1:**.

DELETE

may be used as a command or a statement to delete files on a diskette.

DELETE "testdata"	The file **testdata** is deleted.
DELETE "test*"	All files which begin with **test** are deleted.

RENAME

is used as a command or a statement to change the name of a file.

Example:

RENAME "old","new"	The diskette file with the name **old** is assigned the new name **new**.

PASS

can be used as a command or a statement to send instructions to the disk operating system.

Examples:

PASS "n0:procedurebib,a1"	Formats a new diskette on disk drive **0**. This diskette gets the name **procedurebib** and the identification number **a1**.
PASS "n2:diskname,01",9	Formats a new diskette on the extra disk drive (no. **2**) with unit number **9**.
PASS "v"	Clean house (garbage collection): The files on the diskette are collected and any open files are closed. The letter **v** represents the word **validate**.

Note:
There are additional instructions which can be transferred to the disk operating system using PASS. But there are more suitable COMAL-instructions for accomplishing the same functions.

SELECT INPUT - SELECT OUTPUT

SELECT INPUT

may be used as a command or a statement. It causes subsequent read-in, which normally would occur from the keyboard, to come from the specified sequential ASCII file. This read-in can be terminated by pressing the <STOP> key, by an END-OF-FILE or by errors in the program. At this point input will again be from the keyboard.

INPUT statements, KEY$ and **inkey$** also receive their input from the SELECT INPUT file. The COMAL system interprets this input as if it came from the keyboard and echoes it in the usual manner to the screen.

If SELECT INPUT is used as a command it can be used to redefine the meanings of the function keys.

SELECT INPUT "kb:"	Keyboard input. As with the start up or restart of the COMAL system.
SELECT INPUT "checkfile"	**checkfile** will be read in as if it came directly from the keyboard.

SELECT OUTPUT/SELECT

can be used as a command or as a statement. It is used to select the unit to which subsequent output will be sent. If one simply writes SELECT, the system will automatically add OUTPUT in the program listing after the program has been scanned or run.

SELECT OUTPUT "ds:"	Printout is sent to the screen, as when the computer first is started up.
SELECT OUTPUT "lp:"	Printout is directed to the printer.
SELECT OUTPUT "0:namefile"	A sequential file with the name **namefile** is created on disk drive **0**, and subsequent printout is directed to the file.

Notes:

SELECT OUTPUT can be abbreviated to **SELECT**. The system automatically adds **OUTPUT** after a scan or a run.

Printout will automatically return to the screen after the LIST command has been executed.

Even if printout is directed away from the screen, e.g. to a printer, text provided in INPUT statements will still be directed to the screen.

Commands for System Start Up

BASIC - SYS to COMAL

BASIC

The **BASIC** command directs the computer to initiate the Basic operating system.

The computer can be directed back to the COMAL system with the instruction:

SYS 50000

Both instructions cause all information in working memory to be deleted.

Commands and Statements concerning the Use of Machine Code Program Packages

(See also Chapter 8 on COMAL and programs in machine code.):

USE - LINK - DISCARD

USE

may be used as a command or a statement to append a named machine code program package to the COMAL program in working memory. The name of the package is hereby made known to the COMAL interpreter.

The instruction is used for example to make the built-in packages in the COMAL cartridge accessible in a program. See more about how to use packages in Chapter 5.

Example:

USE graphics	The package **graphics** is activated.

LINK

is a command which fetches a file with a machine code package from diskette and transfers a copy into working memory. The name of the package can then be made known to the program by means of the USE instruction.

Example:

LINK "obj.driver"	The object code file with the name **obj.driver** is fetched.
USE driver	The above LINKed file contains the package with the name **driver**, which is hereby activated.

Note:
A machine code program which is associated with a COMAL program by means of the command **LINK** is saved together with the COMAL program as one file using the SAVE command. A later LOAD will automatically fetch both the COMAL program and the machine code program.

DISCARD

is a command which removes all machine code program packages from working memory.

The COMAL program is not lost, but the interpreters name table is only intact again after a RUN or a SCAN has been performed.

Statements used during Read-In and Printout

INPUT - INPUT AT - KEY$

INPUT

is a statement which reads data into a program during execution. After an INPUT statement the system stops execution and waits for a user response. The cursor flashes at the beginning of the input field. All responses must be terminated by a <RETURN>.

Examples:

INPUT "Total ": number	The system awaits **number** as response.
INPUT "What's your name? ": name$	The system awaits a string input.
INPUT "Position (X,Y) = ": x,y	Several numbers can be entered in the same INPUT statement.
INPUT "Item number: ": no;	A (;) or (,) after the variable name suppresses the carriage return after the answer.

INPUT AT

acts like INPUT with the added possibility of placing the input field anywhere on the 25 lines and 40 columns of the screen.

Examples:

INPUT AT 4,10: "Number = ": no	The input message starts on line 4, column 10.
INPUT AT 4,7,15: "Name ": text$	The input message starts on line 4, column 7. The input field is limited to the 15 following spaces which are protected from other uses.

Special case:

A **0** given as line or column number means **current value**.

Example:

INPUT AT 0,0,10: "Town ":town$ The input message starts at the present line and column, but the response field is limited to 10 characters.

See also INPUT FILE and SELECT INPUT.

KEY$

is a function which reads the keyboard input buffer to determine the last character activated. If no character has been sent, then the function returns the value **chr$(0)** or ""0"". Program execution does not wait in contrast to the INPUT statement and the function **inkey$** (in the **system** package).

Examples of use:

WHILE KEY$=CHR$(0) DO NULL The program 'hangs' in the same line until the user presses any key.

```
DIM answer$ OF 1
PRINT "Answer yes/no"
REPEAT
  answer$:=KEY$
UNTIL answer$ IN "yYnN"
```

The system waits for keys y, Y, n or N to be pressed.

PRINT - PRINT AT - PRINT USING - TAB - ZONE

PRINT

may be used as a command or a statement. It is used to print data on the screen or send it to other output devices. If the PRINT line contains several items, they can be separated by a semicolon (;). This will cause a single space to be printed between each item. If a comma (,) is used, the the number of spaces between the beginning of each item is determined by the ZONE instruction. During program coding PRINT can be abbreviated to ;.

Examples:

PRINT "Result: ";speed;"m/s"	text and numbers can be mixed in the printout.
PRINT	Prints out an empty line.
PRINT text$;	The carriage return is supressed by terminating the PRINT line with a (;) or a (,).

PRINT AT

can be used as a command or as a statement. It makes it possible to print numbers or text at any character position on the screen. Line numbers may range from 1 - 25, and column numbers from 1 - 40.

Example:

PRINT AT 3,12: "Name is"; name$	The printout begins in the 3. line, column 12.

Special case:
A **0** as line or position number means **present** or **current**.

Example:

PRINT AT 0,30: "COMAL"	Print on the present line, column 30.

PRINT USING

can be used as a command or a statement. It is used for printing numbers in a well defined format.

Examples:

PRINT USING "Price ###.##": price	The amount is written in the format determined by the # signs and the decimal point. In this example there is room for 3 digits before the decimal point and 2 digits after it.

The various PRINT options can be combined:

PRINT AT 10,15: USING "Speed = ##.##": speed

Note:
If the number is too big to fit in the specified format, the printout will consist of a row of stars: *****

TAB

is a system function which is used in connection with the PRINT instruction. TAB is an abbreviation for **TAB**ulation.

Example:

PRINT "Itemnumber: ",TAB(25),no — After the text **Itemnumber:** has been printed, the system will move the cursor to column 25 where **no** will be printed.

See also PRINT FILE and SELECT OUTPUT.

ZONE

is a statement and a function which is used in connection with the comma (,). It is used to define the interval between columns in PRINT printouts. When COMAL is initiated and after the use of the command NEW, ZONE is equal to **0**.

Examples:
After start-up:

PRINT 23,56,89	will be printed out as
235689	with no spaces between numbers, because ZONE equals 0.
ZONE 5	The column interval is set to 5.
PRINT 23,56,89	will now be printed out as
23 56 89	

The first number will begin in column 1, the next in column 6, the next in 11, etc.

spacing:=ZONE

ZONE can be used as a function for example to assign a value to the variable **spacing** which is given the current ZONE value.

PAGE - CURSOR

PAGE

can be used as a command or a statement. It is used to clear the screen. If a printer has been selected as the output device, a form feed instruction will be sent to the printer.

CURSOR

can be used as a command or a statement. It can be used to position the cursor on the screen. The character position 1,1 is in the upper left-hand corner, and 25,40 is in the lower right-hand corner.

Examples:

CURSOR 15,30	Place the cursor on line 15, column 30.
CURSOR 0,10	Move the cursor to the present line, column 10. A **0** means **present** or **current**.

Note that the specification of the screen position using CURSOR, INPUT AT and PRINT AT use the line and column method in contrast to high resolution graphics. In graphics the position is specified using a conventional (X,Y) coordinate system.

READ - DATA - RESTORE - Label: - EOD

READ

is a statement which is used to read values from a DATA statement. If the READ statement contains several variable names, then these are separated by commas (,).

Example:
READ name$,street$,no,postno,town$

DATA

is a statement which contains the values which the variable names in a READ statement are assigned. DATA statements are not executed. For this reason they can be placed anywhere in a program. However DATA statements are local within a closed procedure or a closed function.

Examples:
The DATA statement can contain both text and numbers. Text must be enclosed within quotation marks ":

DATA "John Smith","Easton","Pennsylvania"

DATA 230,$e6,%11100110	**DATA** statement can contain both decimal numbers, hexadecimal numbers and binary numbers.

RESTORE

can be used as a command or as a statement. It sets the DATA pointer to point at the first DATA statement in a program or to the first statement right after a **label**.

label:

is a freely chosen name which is used to specify an entry point at some line in the program. The label is not executed like an instruction. It can be used in connection with RESTORE (and GOTO). See the summary example after the definition of EOD.

EOD

is a boolean (logical) system function which is used during a READ from DATA statements. EOD means **End Of Data**. As long as DATA-values remain in the list, EOD is FALSE. When the last DATA-value has been read, then EOD is set to TRUE.

Summary example:

```
DATA "screws",112,"nails",50
toys:
DATA "cars",220,"dolls",35
DATA "balls",76,"jump ropes",24
DIM name$ of 20
RESTORE toys
WHILE NOT EOD
  READ name$,total
  PRINT "There are ";total;name$;"left."
ENDWHILE
```

Notes:

It is usually convenient to place DATA statements near the beginning or the end of the program, so they are easy to find and revise.

A label **toys:** has been placed just before the DATA statements containing the list of toys.

RESTORE toys assures that READ begins in the following line.

Read-in and printout of the toy inventory continues until EOD is set equal to TRUE. This happens when there is no DATA left in the list.

Instructions for Communication with Files

MOUNT - CREATE

MOUNT

can be used as a command or as a statement. It sets up a diskette which has just been placed in the disk drive, getting the diskette ready for reading and writing operations. Cassette tapes do not require this, and diskettes will usually operate properly without being MOUNTed. To be on the safe side it is wise to MOUNT diskettes each time they are put into the drive.

Examples:

MOUNT	disk drive 0 is initialized. (the same as function key 2)
MOUNT "1:"	disk drive 1 is initialized.

CREATE

can be used as a command or as a statement. It creates a random file on diskette. A file can also be created using the OPEN instruction, but communication with the file can be carried out about 10 times faster, when the file has been CREATEd first.

Example:

CREATE "textfile",300,42	A file by the name of **textfile** with **300** records, each **42** characters (bytes) long.

OPEN FILE/OPEN - READ - WRITE - APPEND - RANDOM

OPEN FILE/OPEN

can be used as a command or as a statement. It is used to open access to a file on a peripheral device, e.g. diskette, cassette, printer etc. Several sequential files can be open at the same time with different stream numbers. The term stream number refers to that fact that a data channel is opened to or from the file. It the word FILE is omitted during program coding, the system will automatically add it to the listing after a SCAN or RUN. There are many ways to open files. See Chapter 6 for further information. In the following only a few examples of the use of READ, WRITE, APPEND and RANDOM will be given.

Examples:

OPEN FILE 3,"datafile",WRITE	The file with the name **datafile** and **stream number 3** is opened to **receive** data. Hereafter in the program stream number 3 is reserved for this file, until the file is closed by means of a **CLOSE FILE 3** instruction.
OPEN FILE 7,"cs:names",READ	The cassette file **names** is opened to return **data** to the program. The file is identified by **stream number 7**.
OPEN FILE 15,"data",APPEND	An already existing sequential disk file with the name **data** is opened for **addition** of new data following the existing data on the file. The file is identified by the **stream number 15**.
OPEN FILE 4,"names.usr",WRITE	A sequential file is opened with the classification **usr** instead of **seq**.

OPEN FILE 5,"text",RANDOM 42 The file **text** is opened. RANDOM indicates that it is a **random access file**. Each record will have room for 42 characters (i.e. bytes) on the diskette. 42 bytes will be taken up on the diskette even though the individual records do not use all this room. Access to the records is speeded however, because each record has the same length. The position of each record can be determined when the record number is known.

OPEN FILE 4,"lp:",WRITE A data stream is opened to the printer

PRINT FILE - INPUT FILE

PRINT FILE

can be used as a command or as a statement. It is used for sending data in ASCII-format to a file on diskette, cassette tape or other peripheral. The file must have been previously opened by means of the OPEN instruction. The file is identified by its stream number.

When PRINT FILE is used to send data to a file, the individual data elements are separated by a carriage return <**CR**>, i.e. ASCII-code 13.

A file which has been written to using PRINT FILE can be read using the instruction INPUT FILE.

Examples:

PRINT FILE 2: item$** The value of the variable is written to the sequential file with stream number 2. The printout is terminated by a <**CR**> after **item$. The file is opened using OPEN 2,..,WRITE or APPEND.

PRINT FILE 4,7: name$ The value of the variable is written to the random access file with **stream number 4, record number 7** (opened with RANDOM).

INPUT FILE

is a command or a statement used to read data from a file which has been opened with **OPEN no,name$,READ or RANDOM.** The file must contain data in ASCII format, written with the PRINT FILE instruction.

Examples:

INPUT FILE 2: item$ The value of the variable is read in from the sequential file with stream number 2. The file must have been opened as a READ type.

INPUT FILE 4,7: name$ The value of the variable is read in from file 4, record 7. The file must have been opened as a RANDOM type.

WRITE FILE - READ FILE

WRITE FILE

is a command or a statement which transfers data to a file in compact binary form. The file is sequential, if it is opened as a WRITE or APPEND type; and it is random access, if it has been opened using RANDOM. WRITE FILE is preferable where possible instead of PRINT FILE, because the binary form takes up less space, and access is faster. It is not possible to use WRITE FILE to store data on a cassette tape unit.

Examples:

WRITE FILE 2: first$,last$,tel The values of the variables are written in binary form to the sequential file with stream number 2. The file must have been opened earlier with the instruction OPEN 2,...,WRITE or APPEND.

WRITE FILE 3: tablevalues() The entire set of numbers represented by **tablevalues()** is written to file 3.

WRITE FILE 4,12: no,text$,other$ The values of the variables are written in binary form to a random access file. The stream number is 4, and the record number is 12. The file must have been opened earlier using OPEN 4,...,RANDOM.

READ FILE

is a command or a statement which is used to read data from a file which has previously been opened using the instruction **OPEN no,name$,READ** or **RANDOM**. The file must contain data in binary form, written with the instruction WRITE FILE.

Examples:

READ FILE 2: first$,last$,tel The data values are read in from the sequential file with stream number

2. The file must have been opened as a READ type.

READ FILE 4,12: no,text$,other$ The data values are read in from file no 4, record 12. The file is random access and must have been opened with RANDOM.

CLOSE FILE/CLOSE

can be used as a command or as a statement. It closes files which have been opened with the OPEN instruction. Serious errors can arise if one attempts to copy or rearrange open files. If the word FILE is omitted when this instruction is used as a statement, it will be added automatically by the system after a SCAN or RUN.

Examples:

CLOSE	All open files are closed.
CLOSE FILE 2	The file with **stream number 2** is closed.

EOF

is a logical system function which is used during read-in from a file. EOF means END of FILE. EOF is always used including a stream number: **EOF(<stream number>)**. As long as data elements are remaining in the file, EOF equals FALSE (=0). When last element has been read, EOF equals TRUE (=1).

Example:

```
no:=0
WHILE NOT EOF(2) DO
  no:+1
  READ FILE(2):digit(no)
ENDWHILE
```

Data is read from a file with stream number 2. The read-in terminates, when no elements are left in the file.

UNIT - UNIT$

UNIT

can be used as a command or as a statement. It is used to specify which unit is to be used for file operations when the file name does not contain this information. When COMAL is started, disk drive number **0** is automatically selected as the unit. See Chapter 7 on Peripheral Equipment for further information.

The following units may be selected:

cs: cassette
0: disk drive no 0 (default)
1: disk drive no 1
2: extra disk drive (usual choice)

Note that if a second disk drive is connected via the IEEE serial bus, it should be set up to act as 'device 9'. It will then repond to COMAL instructions when referenced as unit 2.

Example:
UNIT "cs:" Cassette is the default unit.

UNIT$

is a system function which returns the name of the unit to be used, if no other specification is given in the file name.

Example:
PRINT UNIT$ the system responds e.g. with **0:**

Programming Structures

Conditionals
Loop Statements
Error Handling
Procedures and Functions

Conditionals

IF - THEN - ELIF - ELSE - ENDIF

are statements which are used in IF-THEN structures. An IF-THEN statement can be formulated in many different ways. The fundamental principle is, however, quite clear: If a <**logical expression**> is **true**, then the associated statements will be executed. Another way of expressing the same thing is to say that if a given <**condition**> is *fulfilled*, then the associated statements will be executed.

Example 1:

IF<logical expression> THEN <statement>
is a single line version: If the <logical expression> is true, then the

<statement> after THEN is executed. Otherwise the program just continues in the next line.

```
IF answer$="yes" THEN print'data
```

Example 2:

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

Multiline version:
If the expression is true, the statements between THEN and ENDIF are executed. Otherwise execution jumps to the line after ENDIF

```
IF number>=0 THEN
  square'root:=SQR(number)
  PRINT "The square root of";number;"is";square'root
ENDIF
```

Example 3:

```
IF <logical expression> THEN
   ...
  <statements>
   ...
ELSE
   ...
  <statements>
   ...
ENDIF
```

If the expression is true, then the statements between THEN and ELSE are executed. Otherwise the statements between ELSE and ENDIF are executed.

```
IF answer$ IN "aeiou" THEN
  PRINT answer$;"is a vowel."
  PRINT "Want to try again?"
ELSE
  PRINT answer$;"is not a vowel."
  PRINT "The letters: aeiou are vowels,"
  PRINT "all other letters are usually consonants."
ENDIF
```

Example 4:

```
IF <condition1> THEN
    <statement>
   ...
ELIF <condition2> THEN
  <statement>
   ...
  <statement>
   ...
ENDIF
```

ELIF is short for ELSE IF. If **<condition1>** is fulfilled then the statements between THEN and the first ELIF, are carried out. Then program executing continues after ENDIF. If **<condition1>** is not fulfilled, then **<condition2>** i checked. If true, then the statements down to the next ELIF are executed. Next, control passes to the line after ENDIF. Otherwise **<condition3>** is checked, etc.

```
IF number=0 THEN
  add'data
ELIF number=1 THEN
  delete'data
ELIF number=2 THEN
  print'data
ENDIF
```

Example 5:

```
IF <condition1> THEN
  <statement>
  ...
ELIF <condition2> THEN
  <statement>
  ...
ELSE
  <statement>
  ...
  ENDIF
```

If no condition is fulfilled, then the statements between ELSE and ENDIF are executed

```
IF a$="mail" AND b$="box" THEN
    PRINT "Yes indeed!"
    PRINT "The word should be ;a$+b$
ELIF a$="box" AND b$="mail" THEN
    PRINT "Try reversing the words."
ELSE
    PRINT "The words don't agree."
    PRINT "Look at the drawing again,"
    PRINT "and try again!"
ENDIF
```

CASE - OF - WHEN - OTHERWISE - ENDCASE

are statements which are used in **the CASE-structure** to direct program execution in a situation where a number of choices are available.

Example:

```
CASE <expression> OF
WHEN <1st value>
  <statement>
    ...
WHEN <2nd value>
  <statement>
    ...
WHEN <3rd value>
  <statement>
    ...
    .
    .
OTHERWISE (can be left out)
  <statement>
    ...
ENDCASE
```

Example 1:

```
CASE answer OF
WHEN 1
    PRINT "Hm.."
WHEN 2
    draw'line
WHEN 3,4
    draw'polygon
OTHERWISE
    draw'cirkel
ENDCASE
```

Depending on the value of **answer**, one of the procedures will be executed. If the answer is 1,2,3 or 4, then the statements under the corresponding WHEN are executed. Otherwise the statements following OTHERWISE are carried out. The structure always ends with ENDCASE.

Example 2:

CASE works with string constants as well:

```
CASE country$ OF
WHEN "Denmark"
    PRINT "Yes. Correct!"
WHEN "Scandinavia","Sweden","Norway"
    PRINT "Close. More specific, please."
WHEN "Europe"
    PRINT "Go North."
OTHERWISE
    PRINT "Far out.."
ENDCASE
```

Loop Statements

REPEAT - UNTIL

are statements which are used in the **REPEAT-structure**. The statements within the REPEAT-UNTIL loop are repeated until the logical (boolean) expression in the UNTIL statement is true.

Example 1:
REPEAT <statement> UNTIL <logical expression>
is a single line version:
<statement> is executed until <logical expression> is true.

REPEAT read'file UNTIL text$="Susan" OR EOF(no)

The procedure **read'file** will be carried out until the logical expression is true. Either the variables **text$** is equal to "**Susan**", or **EOF(no)** is true (which will occur if there is no more text in the file being read).

Example 2:

```
REPEAT
  <statement>
  ...
UNTIL <logical expression>
```

Multi-line version:
The statements between REPEAT and UNTIL run until the logical expression is true.

```
REPEAT
    INPUT "New number ": a
UNTIL a<0
```

The INPUT statement will be carried out until the rumber read in is negative.

Note that the statements in the REPEAT structure are always carried out at least once, because the logical expression is at the end of the loop.

WHILE - DO - ENDWHILE

are statements which are used in the **WHILE-structure**.

The statements within the WHILE-ENDWHILE loop are repeated as long as the logical expression in the WHILE statement is true.

Example 1:

WHILE <logical expression> DO <statement>
is a single line version:
As long as <logical expression> is true <statement> is executed.

WHILE name$<>"Peter" DO get'name
The call for the procedure **get'name** is repeated, as long as **name$** is different from **"Peter"**.

Example 2:

```
WHILE <expression> DO
  <statement>
  ...
ENDWHILE
```

As long as <**expression**> is true, the statement between DO and ENDWHILE continue to be executed.

```
b:=1
WHILE KEY$=""0"" DO
  b:=2*b
  PRINT 1/b
ENDWHILE
```

As long as no key is pressed, new numbers in the series will continue to be printed out. ""0"" equals CHR$(0).

Notice that the keyword ENDWHILE must not be used in the single line version.

FOR - TO - STEP - DO - ENDFOR

are statements which are used in the **FOR - ENDFOR structure**. The statements within the FOR loop are repeated a predetermined number of times, then program execution continues with the line after ENDFOR. The loop variable <**counter**> **is local.**

Example 1:

FOR <counter>:=<start> **TO** <end> **DO** <statement>
is a single line version:
The loop is repeated <**end**>-<**start**>+**1** times with <**counter**> equal to *<start>*, <**start**>+**1**,..., until <**end**> is passed.

FOR n:=0 TO 30 DO PRINT a(n);

Example 2:

```
FOR <counter>:=<start> TO <end> DO
 <statement>
 ...
ENDFOR <counter>

FOR no:=1 TO 10 DO
  INPUT "Name: ":name$(no)
  INPUT "text: ":text$(no)
ENDFOR no
```

The FOR loop is repeated 10 times with the variable **no** equal to **1, 2,..., 10**

Example 3:
Version with **STEP parameter**:

```
FOR angle:=0 TO 6.3 STEP 0.1 DO
 PRINT COS(angle);SIN(angle)
 PRINT COS(angle)↑2+SIN(angle)↑2
ENDFOR angle
```

As indicated by the STEP parameter, **angle** will take on the values 0, 0.1,..., 6.3

```
FOR i#:=max TO min STEP -1 DO
 moveto(0,0)
 drawto(x(i#),y(i#))
ENDFOR i#
```

The integer variable **i#** increases the speed. The STEP parameter can also be negative.

Note:
The keyword ENDFOR is not used in the single line version.
The single line version can also be used as a command.

LOOP - EXIT - EXIT WHEN - ENDLOOP

are statements which are used in the **LOOP-ENDLOOP structure**. The statements in the LOOP-ENDLOOP segment are repeated until an EXIT or EXIT WHEN statement is executed. Next program execution is continued in the line after ENDLOOP. There can be 0, 1 or more EXIT's in a LOOP-ENDLOOP structure.

Example:

```
LOOP
 <statement>
 ...
 EXIT WHEN <logical expression>
 <statement>
 ...
ENDLOOP
```

```
LOOP
 INPUT "Text ": text$
 EXIT WHEN text$="end"
 WRITE FILE 3: text$
 do'test
ENDLOOP
```

Text is read in, written to file 3 and examined in the procedure **do'test**, until the text **"end"** is read in.

Error Handling

TRAP - HANDLER - ENDTRAP

are statements which are used to control program execution after errors are encountered. If errors occur in the statements between TRAP and HANDLER (called the TRAP part), then the statements between HANDLER and ENDTRAP (the HANDLER part) are executed. Otherwise the program continues with the line after ENDTRAP. In this way one can avoid having the program stop e.g. due to a user data-entry error.

Example:

```
TRAP
  INPUT "No. ": no
HANDLER
  check'error
ENDTRAP
```

If errors occur during read-in, the system will jump down to the HANDLER part and carry out the procedure **check'error**.

ERR - ERRFILE - ERRTEXT$

are system functions which are used in connection with the HANDLER part of the TRAP structure to identify errors. See Appendix F on error numbers and error messages.

ERR contains the error number.
ERRFILE contains the number of a file, if one was in use when a read or write error occurs.
ERRTEXT$ contains the text with the error message.

Example 1:

```
TRAP
  INPUT "Exponent ":exponent
  PRINT 10↑exponent
```

```
HANDLER
  PRINT ERRTEXT$
  CASE ERR OF
  WHEN 2
    PRINT "Exponent too large"
  WHEN 206
    PRINT "Exponent is a number"
  OTHERWISE
    PRINT "Please try again!"
  ENDCASE
ENDTRAP
```

Example 2:

```
TRAP
  INPUT "Filename: ":name$
  OPEN FILE 2,name$,READ
  OPEN FILE 3,"savefile",WRITE
  transfer(name$,"savefile")
HANDLER
  CLOSE
  IF ERRFILE=2 THEN
    PRINT "Error in read-in"
  ELIF ERRFILE=3 THEN
    PRINT "Error during print-out"
  ELSE
    PRINT "Not an input/output error"
  ENDIF
  PRINT ERR;ERRTEXT$
ENDTRAP
```

REPORT

is a command and statement which is used in connection with the **TRAP-structure**. REPORT can be used in several ways to reveal an error and to direct subsequent error handling. REPORT can be used with or without an argument:

REPORT	Repeat earlier error. (only as statement)
REPORT errorno	Report an error with **errorno**.
REPORT errorno,errortext$	Report **errorno** and **errortext$**.

The instruction has various effects according to where it occurs in the structure.

REPORT **outside the TRAP-ENDTRAP** structure:
The error is reported to the system, which will then react to the error.

REPORT **in TRAP part** of the structure:
Program execution is directed to the HANDLER part, where the user program handles the error.

REPORT **in HANDLER part** of the structure:
Program execution is directed to an external HANDLER structure, if

found. Otherwise the error is reported to the system with an error message on the screen.

Example:

```
TRAP
  INPUT "Name: ":name$
  INPUT "Age:   ":age
HANDLER
  IF ERR=2 OR ERR=206 THEN
    age:=0
  ELSE
    REPORT
  ENDIF
ENDTRAP
```

REPORT can sort out errors: If the response to **Age** is not a number, or the number is too large, then **age** is set equal to **0**. Otherwise the error is reported to the system.

GOTO - <Label:>

GOTO

is a statement which causes program execution to continue at a predetermined place. This place is given by a <**Label**>, i.e. a name followed by a colon (:). It is not possible to jump out of a procedure or into a closed program structure using GOTO.

Example:

```
FOR no:=1 TO 10 DO
  READ FILE 2: number
  IF number<1e-37 THEN GOTO too'small
  PRINT 1/number
ENDFOR no
too'small:
PRINT "Divisor too small."
```

<Label:>

is a name which is used to identify a program line. The program line is not executed. Execution continues in the line following <**Label:**>. Labels are used in connection with GOTO and RESTORE.

Examples:

See **GOTO** example.

```
DATA 2,4,5,2,1
twodigit:
DATA 12,34,18,54,22
RESTORE twodigit
WHILE NOT EOD
  READ number(no),
ENDWHILE
```

Read-in of numbers from the DATA statements starts with the number **12** due to the statement **RESTORE twodigit.**

Procedures

PROC - ENDPROC

are statements which are used to form the **PROC-ENDPROC structure**. PROC-ENDPROC surround a number of statements which together form a **procedure**. A procedure is a program module, recognized by a name stated in the procedure heading: **PROC <name>**. The procedure is carried out only if it is called from somewhere else in the program using the same name that appears in the PROC heading.

COMAL programs should be created using procedures. In their simplest form, they can be used to break a larger program down into smaller, easy to handle units. More advanced uses with parameter transfer and use of the options REF, CLOSED, IMPORT and EXTERNAL make procedures a programming tool of substantial value.

Example 1:

```
// MAIN PROGRAM
 <statement>
 ....
 <name1>
 <statement>
 ....
 <name2>
 <statement>
 ....
 <name1>
 <statement>
 ....
 ....
 END // MAIN PROGRAM

 PROC <name1>
 <statement>
 ...
 ...
 ENDPROC <name1>
```

```
PROC <name2>
<statement>
...
...
ENDPROC <name2>
```

The statements of the procedure are enclosed in **PROC <name>** and **ENDPROC <name>**. The procedure can be called "by name" from various places in the main program.

```
// MAIN PROGRAM
start'up
read'in
```

The main program consists of program lines, each of which calls a procedure.

```
PROC start'up
  USE system
  textcolors(0,2,1)
  DIM number(10)
  PAGE
ENDPROC start'up

PROC read'in
  FOR no:=1 TO 10 DO
    PRINT "Read in age (",no,") ",
    INPUT "": number(no)
  ENDFOR no
ENDPROC read'in
```

Example 2:

```
<statement>
...
print'out(member,age,name$)
<statement>
...
...
PROC print'out(no,years,text$)
  PRINT
  PRINT "Membership number: ",no
  PRINT "Age               : ",years
  PRINT "Name              : ",text$
ENDPROC print'out
```

Notes on example 2:
In the main program the procedure **print'out** is called. Those values which are contained in the actual parameters **member**, **age** and **name$**, are transferred to the formal parameters **no**, **years** and **text$**, which occur in the procedure heading.

The variable names of the formal parameters are local within the procedure **print'out**.

This form for value transfer is one-way: Values can be passed into the procedure but not from it.

Notes on procedures:
When a procedure has been RUN or SCANned, it can be used as a command.

A procedure can call another procedure, or it can even call itself.

A procedure can be placed within another procedure and thereby be made local for just this procedure. (Similarly, a function and a label will be local within a procedure/function.)

The command **SETEXEC+** will cause every procedure call in the listing to begin with the word **EXEC** (for "execute"). See SETEXEC.

REF - CLOSED - IMPORT

REF

is a parameter type which is used in a procedure call. A REF preceding a parameter in the procedure heading indicates that the name will only be synonomous with the corresponding name in the procedure call. It is called by reference. No room is reserved in the computer's working memory for a new name and value. The value receives only a new, temporary name. Both names **refer** to the same value. In this way room is saved in storage, execution speed is increased, and parameter values can be passed both ways: into and out of the procedure.

Example:

```
<statement>
....
read'ln(class,name$())
<statement>
....
PROC read'ln(REF no,REF a$())
  INPUT "Which class: ": no
  PRINT "Write student names."
  i:=0
  REPEAT
    i:+1
    INPUT "Name: ":a$(i)
  UNTIL a$(i)=""
ENDPROC read'ln
```

While the procedure **read'ln** is carried out, the names **class** and **no** will refer to the same value because of the REF in front of **no**. The same is true for the names **name$** and **a$**. Both refer to the string values in a one-dimensional array.

CLOSED

is an instruction which is used to declare all variable names in a procedure as local. Thus the procedure is 'closed off' from the rest of the program except for transfer of parameter values in the parentheses of the procedure heading. In this way mixing and name conflicts between procedure names and variable names in the rest of the program can be avoided. For example a name can be used locally in the procedure without disturbing the value of a variable with the same name outside the procedure.

Example:

```
a:=10
DIM b(a)
FOR no:=1 TO a DO INPUT "Next number: ": b(no)
minmax(a,b(),min,max)
PRINT min;max
...
PROC minmax(n,a(),REF b,REF c) CLOSED
  b:=a(1);c:=a(1)
  FOR l#:=2 TO n DO
    IF a(l#)<b THEN b:=a(l#)
    IF a(l#)>c THEN c:=a(l#)
  ENDFOR l#
ENDPROC minmax
```

The procedure **minmax** is CLOSED so that it can be used without worrying about the names of the variables in the procedure.

IMPORT

is a statement which is used in closed procedures to bring in variables, procedures and functions from outside the procedure. In this way they can be made accessible for use in an otherwise closed procedure.

Example:

```
<statement>
...
print'out(points())
<statement>
...
PROC print'out(number()) CLOSED
  IMPORT total, t(), sort
  DIM prod(total)
  FOR no#:=1 TO total DO
    prod(no#):=number(no#)*t(no#)
    PRINT no#;proc(no#)
  ENDFOR no#
  sort(number(),total)
  sort(t(),total)
  FOR no#:=1 TO total DO
    PRINT no#;number(no#)*t(no#)
```

```
  ENDFOR
ENDPROC print'out
```

Even though the procedure **print'out** is closed, the variable **total**, the table **t()** and the procedure **sort** are made accessible by means of the **IMPORT** statement.

EXTERNAL - MAIN

EXTERNAL

is a keyword which is used to indicate that a given procedure is an **external procedure** which must be fetched from the diskette when it is to be used in the program. When creating a procedure for use as an EXTERNAL procedure, it must be closed using the CLOSED instruction and saved using the command SAVE. The SAVEd procedure can be fetched from the diskette later for use in another program, provided it is declared to be EXTERNAL in this program. In this way it is possible to build up a library of procedures. The procedures can then be fetched into the working memory as need for use in programs.

Example:

```
PROC test(a,b$,REF check) CLOSED
  IF a=0 AND b$ IN "abcd" THEN check:=TRUE
ENDPROC test
```

The procedure **test** is CLOSED and SAVEd on diskette with the command **SAVE test "ext.test"**.

It can be used later in another program.

```
// Program start
<statement>
...
test(no,text$,error)
<statement>
...
PROC test(no,text$,REF error) EXTERNAL "ext.test"
// Program end
```

This program will fetch the procedure **test** from diskette, use it and "forget" it again.

The line with the EXTERNAL declaration can be placed anywhere in the program.

MAIN

is a command which is used to bring the system back to the main program, if it should stop during the execution of an EXTERNAL procedure. If execution is stopped in an external procedure, LIST and other editing instructions will work only on the external procedure, until MAIN removes it and brings back the main program.

Functions

FUNC - ENDFUNC - RETURN

are statements which are used in the **FUNC-ENDFUNC structure**. This structure consists of a number of statements which together compose a **user-defined function**. Functions must be introduced with **FUNC <name>** and terminated by **ENDFUNC <name>**. The value which the function returns must be given in a **RETURN**-statement.

Functions can be real functions, integer functions or string functions. A function is computed only if it is called somewhere in the program by the same name which is indicated in the function heading (**FUNC <name>**).

Functions can be associated with the same properties which were available for procedures: **REF, CLOSED, IMPORT** and **(<parameter list>)**. See also under these keywords in Chapter 4. In addition you will find that functions are used in Chapter 3 and in Appendices C and E.

In particular, all functions (after structure check caused by SCAN or RUN) can be called as direct commands.

Example 1:

```
// Main program
// real function
 <statements>
PRINT average(a,b)
 <statements>

FUNC average(x,y)
  RETURN (x+y)/2
ENDFUNC average
```

Example 2:

```
// Main program
// integer function
 <statements>
first#:=vowels#("COMAL")
second#:=vowels#("and functions")
 <statements>
```

```
FUNC vowels#(text$) CLOSED
  number#:=0
  FOR i#:=1 TO LEN(text$) DO
    IF text$(i#:i#) IN "aeiouAEIOU" THEN number#:+1
  ENDFOR i#
  RETURN number#
ENDFUNC vowels
```

Example 3:

```
// Main program
// string function
 <statements>
PRINT mystical$("secret")
 <statements>

FUNC mystical$(a$)
  double:= 2*LEN(a$)
  DIM b$ OF 1, c$ OF double
  c$:=a$
  FOR i:=1 TO double STEP 2 DO
    b$:=CHR$(RND(65,93))
    c$:=c$(:i)+b$+c$(i+1:)
  ENDFOR i
  RETURN c$
ENDFUNC mystical$
```

Example 4:

```
PRINT grab$(0,"Once upon a time")

FUNC grab$(first,a$)
  length:=LEN(a$)
  IF length>1 THEN
    IF first THEN
      RETURN a$(2:)
    ELSE
      RETURN a$(:length-1)
    ENDIF
  ELSE
    RETURN ""
  ENDIF
ENDFUNC grab$
```

If **first**<>0 then the function **grab$** returns the word in variable **a$** except for the first letter, which is grabbed. If **first**=0, then the last letter is grabbed.

Other Functions

ABS - INT - SGN - SQR - PI

ABS

is a function which calculates the absolute value of an expression. It is sometimes called the numerical value. If the numerical value of the expression is negative, the sign is changed to positive. A positive value remains unchanged.

Examples:

ABS(3.25)	equals 3.25
ABS(-7.46)	equals 7.46
ABS(x-7)	the result depends on the value of **x**. (equals x-7 if x>=7; 7-x if x<7)

INT

is a function which calculates the integer part of the value of an expression, i.e. the largest integer (whole number) which is less than or equal to the value of the given expression.

Examples:

INT(3.25)	equals 3
INT(-7.46)	equals -8
INT(1/2)	equals 0

SGN

is a function which assumes the value +1, 0 or -1, when the value of a given expression is positive, zero or negative respectively.

Examples:

SGN(327.54)	equals **=1**
SGN(-45.7)	equals **-1**
SGN(0)	equals **0**
SGN(x/7-y)	the result depends on **x** and **y**.

SQR

is a function which returns the square root. The argument must be non negative (i.e. positive or zero).

Examples:

SQR(16)	equals 4
SQR(4.9e+09)	equals 70000
SQR(x↑2=y↑2)	the result depends on **x** and **y**.

PI

is a system constant which is assigned the value 3.14159266. PI is particularly useful in connection with the use of angles in radian measure, where PI radians corresponds to 180 degrees.

COS - SIN - TAN - ATN

COS

is a function which calculates the cosine of a number. This number must be expressed in radians.

---

x degrees = x*PI/180 radians
x radians = x*180/PI degrees

Examples:

COS(PI/2)	equals 0
COS(2.5)	equals -0.801143616
COS(v*PI/180)	the result depends on the value of v.

SIN

is a function which calculates the sine of a number. This number must be expressed in radians. See under COS.

Examples:

SIN(PI/6)	equals 0.5
SIN(angle)	the result depends on the value of **angle**.

TAN

is a function which calculates the tangent of a number. This number must be expressed in radians. See under COS.

Examples:

TAN(-PI/4)	equals -1
TAN(1.8)	equals -4.28626168

ATN

is a function which calculates the **arc-tangent** (inverse tangent) of a number. The result is a number, expressed in radians.

Examples:

ATN(1)	equals 0.785398163 (PI/4)
ATN(-200)	equals -1.56579637

LOG - EXP

LOG

is a function which calculates the natural logarithm of a positive number. LOG represents logarithms to the base **e**, where **e** is equal to 2.71828183. LOG is the inverse function of EXP.

Examples:

LOG(1)	equals 0
LOG(10)	equals 2.30258509
LOG(-2)	is not defined
LOG(EXP(x))	equals x

EXP

represents the exponential function. EXP(x) = e raised to the x'th power, where **e** is the base of the natural logarithms. EXP is the inverse function to LOG.

e = 2.71828183 to good approximation.

Examples:

EXP(1)	equals 2.71828183 (= e)
EXP(3)	equals **e cubed** = 20.0855369
EXP(t-a*.2)	the result depends on t and a.
EXP(LOG(x))	equals x

CHR$ - STR$ - SPC$

CHR$

is a string function which equals the character which corresponds to the ASCII code of the argument. The opposite operation is performed with the function ORD.

See Appendix A for Commodore ASCII codes.

Examples:

CHR$(65)	equals the character **a**
CHR$(147)	equals the code for **clear screen**
CHR$(<value>)	the result depends on **value**
CHR$(ORD("B"))	equals the character **B**

STR$

is a string function which converts a numerical expression to a string. The reverse operation is performed by the function **VAL**.

Examples:

STR$(1.34)	equals the string "1.34"
STR$(2-5)	equals the string "-3"
STR$(VAL("7"))	equals the string "7"

SPC$

is a string function which returns the specified number of spaces ("blanks").

Examples:

PRINT "1",SPC$(10),"2"	**10 spaces** are printed between **1** and **2**.
text$:= "a"+SPC$(8)+"jk"	**text$** is set equal to **"a jk"**
blanks$:=SPC$(LEN(name$))	**blanks$** is a string with the same number of spaces as there are letters in **name$**.

ORD - VAL - LEN

ORD

is a function. The value of ORD is the ASCII value of det first charcter in the string argument. The "reverse" operation can be carried out by the function CHR$.

See Appendix A for Commodore ASCII codes.

Examples:

ORD("F")	equals 198
ORD("doors")	equals 68
ORD(by$)	the result depends on by$
ORD(CHR$(8))	equals 8

VAL

is a function which transforms a legal string argument to its corresponding numerical value. To be legal the string must be composed of the digits 0,...9, the signs + = —, decimal point . or **e** used to specify exponential notation. The reverse operation is carried out with the function **STR$**.

Hexadecimal and binary notation is permitted.

Examples:

VAL("123")	equals the number 123
VAL("2"+"3")	equals the number 23
VAL("4e12")	equals the number 4e+12
VAL("abe")	illegal
VAL(STR$(2))	equals the number 2
VAL("$fe")	equals the number 254

LEN

is a function, whose value is the length of the string argument.

Examples:

LEN("abcd")	equals the number 4
LEN(name$)	the result depends on name$
LEN("")	equals the number 0
LEN("a ki")	equals the number 5

TRUE - FALSE

TRUE

is a system constant which always equals **1**.

FALSE

is a system constant which always equals **0**.

TIME

is a command, statement and function used with the system's built-in real-time clock.

The clock measures time in jiffies.

---

1 second = 60 jiffies.
1 day   = 5184000 jiffies
(The clock is reset to zero.)

TIME can be used to set the clock or to read the time since the previous zeroing.

Examples:

TIME 0	The clock is zeroed.
TIME 3600	The clock is set to 3600 jiffies, i.e. 1 minute.
sec:=INT(TIME/60)	**sec** is set equal to the number of seconds since the last zeroing.

RANDOMIZE - RND

RANDOMIZE

is a command and statement which is used to place the random number generator at an arbitrary point in the random number series. The random numbers are created with the function **RND**.

Examples:

RANDOMIZE	The initial placement in the number series is determined by the time interval since the last TIME operation. Since the number of jiffies (1/60 sec) will generally be quite random, a really random sequence can be assured.
RANDOMIZE 6	If **RANDOMIZE** is followed by a number, this number will indicate the starting position in the random sequence each time random numbers are generated. This will cause the same sequence to be generated when RND is used.

RND

is a function which selects a random real number from a random number sequence of evenly distributed 'random' numbers.

RANDOMIZE is used to position the random number generator at an arbitrary position (based on the clock) in this series.

Examples:

number:=RND	An arbitrary real number between 0 and 1 is chosen: 0<=RND<**1.**
no:=RND(-10,30)	A random number chosen among -10,-9, ..,29,30 is selected.
PRINT RND(min,max)	A random integer between **min** and **max** (inclusive) is printed out.

ESC - TRAP ESC

are keywords which control the action of the <**STOP**> key.

ESC is a system function. Its value depends on whether the statement **TRAP ESC+** or the statement **TRAP ESC-** has been executed:

If TRAP ESC+ has been executed (it is the default condition), then pressing the <**STOP**> key will interrupt program execution. The ESC function has no meaning.

If TRAP ESC- has been executed, then pressing <**STOP**> will NOT interrupt the program. ESC will have the value FALSE, until <**STOP**> is pressed. Then it will remain TRUE until the value of ESC is read in the program.

Sample sequence:

TRAP ESC-	The <**STOP**> key will now not stop the program and ESC is assigned the value FALSE.
<STOP> is pressed	ESC is set equal to TRUE.
dummy:=ESC	ESC is reset to FALSE.
TRAP ESC+	The <**STOP**> key regains its usual function.

Operators

See Appendix C for a more detailed treatment of operators.

DIV - MOD

DIV

is an operator which yields the value of the integer part of the quotient after division. **x DIV y** is the same as **INT(x/y).**

Examples:

5 DIV 2	equals 2
74 DIV 10	equals 7
(x+3) DIV y	the result depends on x and y.

MOD

is an operator which computes **the remainder** after division. **x MOD y** is the same as **x-INT(x/y)*y**.

Examples:

5 MOD 2	equals 1
74 MOD 10	equals 4
8.25 MOD 2.1	equals 1.95
(4-x) MOD z	the result depends on x and z.

Logical Operators

NOT - AND - AND THEN - OR - OR ELSE

NOT

is a logical operator which changes the truth value of an expression.

Truth table:

a	NOT a
TRUE	FALSE
FALSE	TRUE

Examples:

```
WHILE NOT EOF(2) DO
  READ FILE 2: number
  PRINT number;
ENDWHILE
```

The loop continues until there is no more data in the file with stream number 2.

```
IF NOT ok THEN read'status(ok)
```

The procedure **read'status** is executed until the variable **ok** becomes TRUE (**<>0**).

AND

is a logical operator which determines the truth value of a combined expression, **a AND b**. The combined expression is only TRUE, if both **a** and **b** are true.

Truth table:

a	b	a AND b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Examples:

7=2 AND 3=3 gives the value FALSE

WHILE expression1 AND expression2 DO make'drawing
If both **expression1** and **expression2** are TRUE, then the procedure **make'drawing** is executed. Otherwise it is not.

AND THEN

is a logical operator which is an extension of the operator AND: **a AND THEN b**. The same rules apply to AND THEN as for AND; but if the first expression **a** is false, the expression **b** is not computed, for it is certain that the entire expression will be FALSE.

Example:

```
a$:="test";I:=1
length:=LEN(a$)
WHILE I<=length AND THEN a$(I)<>"." DO I:+1
```

For i:=5 an error will occur in the logical expression **a$(I)<>"."**, if this case is not eliminated by the first condition.

OR

is a logical operator which determines the truth value of a combined expression, **a OR b**. The combined expression is true, if just one of the expressions **a** or **b** is TRUE.

Truth table:

a	b	a OR b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Examples:
7=2 OR 3=3 gives the value TRUE.

```
REPEAT
 <statement>
  ...
UNTIL no>4 OR ans$ IN "yY"
```

The statements in the REPEAT-loop are repeated until **no>4** or **ans$** is a **y** or a **Y**.

OR ELSE

is a logical operator which is an extension of the operator OR: **a OR ELSE b**. The same rules apply for OR ELSE as for OR; but if the first expression **a** is true, then the expression **b** is not calculated, since the combined expression must be TRUE.

Example:
IF a#=0 OR ELSE b/a#>100 THEN new'problem

If **a#** equals 0, then the first logical expression is true. In this case an evaluation of the last expression (involving an illegal division) is superfluous.

IN

is a operator which returns the position of a search string in a given text: **string IN text**.

The value is the number in the text of the first character in the search string. If the search string is not found, then the value 0 is returned.

IN can therefore be used for example to determine if a response is contained in a string containing acceptable answers.

Examples:

x:="gram" IN "programing"	**x** gets the value 4.
PRINT "mel" IN "Comal program"	**0** is printed.
IF answer$ IN "nN" THEN STOP	If **answer$** consists of the letter **n** or **N**, the expression is TRUE, and the program stops.

Special example:

If the search string is empty, i.e. equal to "", then IN returns **the text length + 1**.

x:="" IN "Comal for CBM" **x** = 14.

BITAND - BITOR - BITXOR

BITAND

is a logical (boolean) operator which executes an AND on each bit in the binary representation of two numbers: **a BITAND b**.

All numbers which are to be compared with the operators BITAND, BITOR or BITXOR must be integers in the interval 0-65535, i.e. binary numbers between %0000000000000000 and %1111111111111111.

Rules:

BITAND a	b 00	01	10	11
00	00	00	00	00
01	00	01	00	01
10	00	00	10	10
11	00	01	10	11

E.g. %	1	0	0
	AND	AND	AND
%	1	1	0
%	1	0	0

Examples:

%0011 BITAND %0101 gives %0001 (decimal 1)
17 BITAND 18 gives 16
$fe BITAND 5 gives 4

IF PEEK(userport) BITAND %1100 THEN register

If the contents of memory address **userport** has the bit pattern **%00001100**, then the procedure **register** will be executed.

BITOR

is a logical (boolean) operator which executes an OR on each bit of the binary representation of two numbers: **a BITOR b**.

Rules:

BITOR a	b 00	01	10	11
00	00	01	10	11
01	01	01	11	11
10	10	11	10	11
11	11	11	11	11

E.g. %	1	0	1
	OR	OR	OR
%	1	0	0
%	1	0	1

Examples:

%1010 BITOR %0110 gives %1110 (decimal 14)
23 BITOR $1b gives 31

BITXOR

is a logical (boolean) operator which executes an XOR (i.e. an "exclusive OR") on each bit in the binary representation of two numbers: **a BITOR b.**

Rules:

BITXOR a	b 00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

E.g.			
%	1	0	1
	XOR	XOR	XOR
%	1	0	0
%	0	0	1

Examples:

%0011 BITXOR %1010 gives %1001 (decimal 9)
17 BITXOR 8 gives 25

Other Instructions

//

is a statement which allows the inclusion of comments in a program. The comment statement is not executed, but is used in the program to clarify its function. Comments make it easier for other programmers (or yourself) who examine the program later to understand how it works.

The comment lines take up room in the working memory but do not slow down a program's execution.

Examples:

```
// graphics window cleared

a$:=b$(1)+b$(LEN(b$)) // a$=b$'s first and last character
```

TRACE

is a command which is used to trace active procedure or function calls. **TRACE** can be used to help find the cause of an error in a program.

Example:

A program might be stopped in a procedure in line 740 due to an error:

```
TRACE
the program stopped at
0740 a$:=character$(1:3)
within
```

0700 PROC print'out(no,character$)
which is called at
0030 print'out(2,"k")

DIM

is a command and statement which is used to **reserve room** in working memory for **arrays** containing numbers or text.

As a statement it will usually occur in the beginning of a program to dimension global indexed variables, but it can also be used locally within a closed procedure.

Arrays with numbers:

DIM table(50)	The array can contain real numbers with indices 1, 2,..,50.
DIM x#(20),y(20)	A DIM-statement can contain several arrays, separated by commas (,).
DIM point(-10:20)	Array with index -10,-9, ..,0,..,20
DIM space(10,40,40)	Three dimensional array
DIM price(0:100,5:10)	Two-dimensional array with indices 0,..,100 and 5,..,10

Note:
If the array specification in the DIM statement does not include a lower index limit, it is automatically set equal to 1.

When created by a DIM statement, all array values are set equal to 0.

String arrays:

DIM name$ OF 30	Room is reserved for 30 characters in the string **name$**.
DIM item$(10) OF 20	Room for up to 10 **item$**-names. Each name may contain up to 20 characters.
DIM text$(0:10,2:5) OF 80	**text$** is a two-dimensional array of words of maximum 80 characters.

Note:
The first time a string is assigned a value, room is reserved in memory for 40 characters, if not previously declared by a DIM statement.

Once dimensioned a string is set equal to the empty string, "".

PEEK - POKE

PEEK

is a function which fetches the contents of a given storage address. The result is an integer between 0 and 255. A "map" with an overview of the use and availability of Commodore 64 memory addresses can be seen in Chapter 8 on Machine Language.

Examples:

line:=PEEK(214)	The line number on which the cursor is currently located is fetched from memory location 214 and the variable **line** is assigned this value.
PRINT PEEK($dd00)	Prints the contents of the parallel port.

POKE

is a command and a statement which is used to place a number directly into a storage address: **POKE address,number**.

You must be careful when using POKE, since sending wrong numbers to random addresses can do strange things to your program. If the worst comes to the worst, it may be necessary to power-down and power-up again to continue programming!

Examples:

POKE 198,0	The counter of the keyboard buffer is zeroed. I.e. the buffer is emptied.
POKE $dd03,%11110000	The direction register of the parallel port has the hexadecimal address $dd03. This address will contain the binary number %11110000 which sets bit 0-3 to inputs and bits 4-7 to outputs.

SYS

is a command and statement which directs program execution to a machine code subroutine starting at the address specified.

Example:

SYS 4000	execute the machine code routine starting at (decimal) address 4000.
SYS 50000	The system carries out a COMAL start-up (this is usually done directly from Basic to start COMAL).

NULL
is a command or statement which is used to do **nothing**! In fact it is quite useful when creating pauses and other situations, where it is desired that the program be delayed until some event (say pressing a key) causes execution to proceed.

Examples:
FOR pause:=1 TO 1000 DO NULL

WHILE KEY$=CHR$(0) DO NULL

STOP - END

STOP
is a statement which is used to stop the execution of a program.
STOP can be placed anywhere in a program, and there can be several STOP-statements in a program. After the program has been stopped, the values of any variables can be examined and/or changed. Using the command **CON** the program can be caused to continue at the line following the STOP statement. However no changes in program syntax may be made.

Examples:

STOP	The program stops with the message: **STOP at xxxx**
STOP "printout finished"	The program stops with the message: **printout finished**.

END
is a statement which completely terminates program execution and marks the conclusion of a program. **END** can be placed anywhere in a program. In contrast to STOP, the program can't be continued with the CON command.

Examples:

END	The program is terminated with the message: **END at xxxx**
END "All finished!"	The program is terminated with the message: **All finished!**

Chapter 5

COMAL Packages

WHat is a package?

In your COMAL cartridge there are 11 program packages with useful procedures. The packages are written in machine code for speed and compactness. They can help you to take full advantage of the many resources available in COMAL and the Commodore 64.

A package and its built-in procedures and functions is made accessible with the command or statement:

USE <package name>

where **package name** is one of the 11 names which follow:
When a package has been activated, its procedures and functions are called by name just as the ordinary COMAL procedures and functions which the user can create. All package procedures can be used as commands as well as program statements. More than one package can be activated at a time.

Overview of packages:

1. **english** , English error messages
2. **dansk** , Danish error messages

3. **graphics** , procedures for X-Y graphics
4. **turtle** , procedures for turtle (Logo) graphics
5. **sprites** , procedures for handling sprites
6. **sound** , procedures for controlling the SID sound chip
7. **system** , procedures for altering system configuration
8. **font** , procedures for defining new character sets
9. **paddles** , a procedure for reading the paddle inputs
10. **joysticks** , a procedure for reading joystick inputs
11. **lightpen** , procedures for control of a light pen

The English Package

USE english activates this package. When activated, all COMAL error messages will be in English. When COMAL is started up, the command **USE english** is executed automatically. This package contains no procedures.

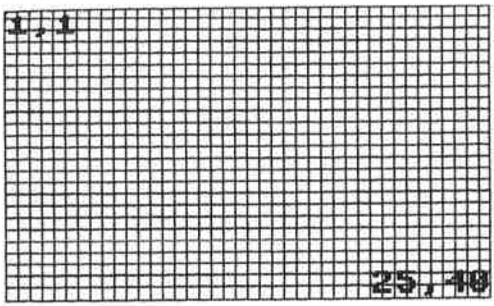
The Danish Package

USE dansk activates the package. All COMAL error messages will then be issued in Danish. The package contains no procedures.

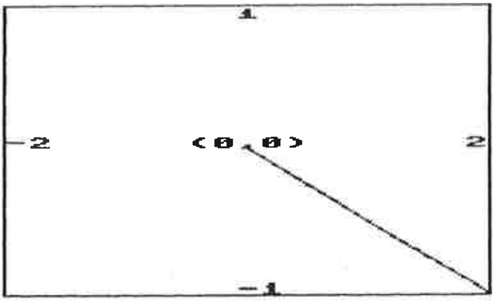
Graphics with COMAL

With the Commmodore 64 you can work with two different display screens: A text screen and a graphics screen.

To work with these screens you can imagine that the computer has two internal 'maps' which show the current state of each of these graphics screens. Only one of these maps can be shown on the display screen at a time.



Normally you will be looking at the text screen. It consists of 25 lines, each with room for 40 characters. Position 1,1 is in the upper left-hand corner, and position 25,40 is at the lower right on your display screen. Thus the text screen has a total of 25 x 40 = 1000 different character locations. In each position a letter, number or graphics character can be placed.



The graphics screen consists of 320x200 = 64000 dots: 320 horizontally and 200 vertically. The dots are identified in a coordinate system by means of a pair of numbers (X,Y). The point (0,0) on the physical display is located in the lower left-hand corner, and the point with coordinates (319,199) is in the upper right-hand corner. Each of these dots is sometimes referred to as a *pixel* (picture element).

The procedures and functions which are used to draw on the graphics screen are made accessible when you use the instruction:

USE graphics or **USE turtle**.

When using the high resolution graphics screen, two further options are available:

graphicscreen(0) , high resolution graphics
graphicscreen(1) , multicolor graphics

Both instructions make the graphics screen visible on the display and the text screen is hidden from view but available for later use. The difference between the two types of graphics display has to do with the number of possible color combinations which can be displayed. See the more detailed discussion of the **graphicscreen** instruction for further information about this.

Use high resolution if you want to make drawings with lots of detail using just one color besides the background color.

If the use of several colors is more important than details, then the multicolor graphics option is the one to use.

A program which draws a yellow border around the display screen might look like this:

```
USE graphics
graphicscreen(1)
pencolor(7)
drawto(319,0)
drawto(319,199)
drawto(0,199)
drawto(0,0)
WHILE KEY$=CHR$(0) DO NULL
```

The last line of the program keeps the graphics screen visible until any key is pressed. When a key is pressed, the condition KEY$ = CHR$(0) will no longer be fulfilled, and the program will end. The computer then displays the text screen, hiding the graphics screen.

After the instruction **USE graphics** has been executed, you can use the function keys <**f1**> and <**f5**> to choose which of the graphics screens you wish to view:

<**f1**> , displays the text screen
<**f5**> , shows the graphics screen

The function key <**f3**> can still be used to issue the command **USE turtle**, causing a split screen to be displayed:

<**f3**> , split screen: graphics screen with 4 lines scrolling text at the top

Pressing <CTRL-u> toggles the effect of <f1>, <f3> and <f5> between text - and graphics mode.

While using COMAL graphics you are not limited to the use of coordinates in the range from (0,0) to (319,199). You can superimpose your own coordinate system onto the graphics screen by using the instruction **window**. All graphics instructions except for the instruction **viewport** and the **sprite** instructions will then be referred to your coordinate system.

Program example:

```
USE graphics
graphicscreen(1)
window(-2,2,-1,1)
moveto(0,0)
drawto(2,-1)
WHILE KEY$=CHR$(0) DO NULL
```

The instruction **window(-2,2,-1,1)** superimposes a coordinate system onto the display screen. The point (-2,-1) is now at the lower left-hand side of the screen, and (2,1) is at the upper right-hand corner.

When high resolution graphics is started up using the instruction **USE graphics**, the coordinate system selected corresponds to the instruction **window(0,319,0,199)**, in accord with the standard screen coordinates. The instruction **USE turtle** performs an automatic **window(-160,159, -100,99)**, so that the origin (0,0) is at the center of the display screen.

If you want to write text on the graphics screen, you can use the special writing instruction **plottext**.

Example:

```
USE graphics
graphicscreen(1)
plottext(0,100,"COMAL graphics")
WHILE KEY$=CHR$(0) DO NULL
```

In Chapter 3 there are further examples of the use of graphics procedures. In addition you will find many examples of the use of graphics on the demonstration diskette (or cassette tape) which accompanied your COMAL cartridge.

Graphics Overview

The packages **graphics** and **turtle** contain the following instructions:

Definition of working area:
viewport - window
Choice of graphics screen and color graphics state:
graphicscreen
Choice of screen:
textscreen - fullscreen - splitscreen
Clearing of graphics screen:
clearscreen - clear
Color choice:
textcolor - textbackground - textborder
pencolor - background - border
(X,Y) graphics:
plot
drawto - moveto
draw - move
setxy
circle - arc
xcor - ycor
Intelligent color fill:
fill - paint

Turtle graphics:
showturtle - hideturtle
turtlesize
home
setheading - heading
penup - pendown
left - right
forward - back
arcl - arcr
Text on the graphics screen:
textstyle - plottext
Information on graphics modes:
inq
Storage and printing of the graphics image:
savescreen - loadscreen
printscreen

In addition it is possible to use the following procedure abbrevations when the **turtle** package is activated:

bk	=	**back**
bg	=	**background**
cs	=	**clearscreen**
fd	=	**forward**
ht	=	**hideturtle**
lt	=	**left**
pc	=	**pencolor**
pd	=	**pendown**
pu	=	**penup**
rt	=	**right**
seth	=	**setheading**
st	=	**showturtle**
textbg	=	**textbackground**

In Depth Look at Graphics Instructions

viewport(<vxmin>,<vxmax>,<vymin>,<vymax>)

is a procedure which limits the area of the display screen in which one can define a coordinate system and draw.

The parameters <vxmin>, <vxmax>, <vymin> and <vymax> always refer to the physical display screen itself with (0,0) in the lower, left-hand corner and (319,199) in the upper, right-hand corner. Note that this procedure is independent of any other coordinate system which may have been chosen using the **window** procedure.

Example:

viewport(0,159,0,99)	It is not possible to draw outside the lower left quadrant of the display screen.

window(<wxmin>,<wxmax>,<wymin>,<wymax>)

is a procedure which defines the coordinate system in the given viewport. The pixel in the lower, left-hand corner of the viewport is assigned the coordinates (<wxmin>,<wymin>). The pixel in the upper, right-hand corner is assigned the coordinates (<wxmax>,<wymax>). All subsequent graphics instructions (except **viewport** and the sprite commands) will refer to this coordinate system until a new one is defined.

On start-up with **USE graphics** the **viewport** is the entire display screen and the coordinate system is defined by **window(0,319,0,199)**

On start-up with **USE turtle** the **viewport** is the entire display screen and the coordinate system is defined by **window(-160,159,-100,99)**. Thus the point (0,0) is in the middle of the display screen.

Example:
window(-1000,2000,-100,200)

graphicscreen(<mode>)

is a procedure which makes the graphics screen appear on the display screen and makes the the text screen invisible.

The graphics screen can be made accessible in two different modes:

graphicscreen(0) , high resolution graphics
graphicscreen(1) , multi-color graphics

The difference between the two modes lies in the manner in which color is handled. The pixels of the display screen are not independent when using color:

In mode 0 (high-resolution graphics) the points of the display are associated in blocks of 64 pixels: (8 on each side). Within each block there may only be two different colors, one of which is the background color. If one attempts to give a pixel in the block a third color, then the entire block will get this color.

In mode 1 (multi-color graphics) resolution in the horizontal direction is not as good, for the pixels are associated in pairs. This means that each block consists of 4 x 8 pairs. Each of these pairs can be assigned a color. If one of the elements of the pair is assigned a color, the other dot will automatically acquire the same color. Within each block four different colors can be displayed at the same time. One of them is the background color. If one attempts to introduce a fifth color, the fourth color will also be given the new color.

textscreen

is a procedure which makes the text screen appear on the display screen. The graphics screen is not visible but still available in computer memory.

It can be necessary in a program to switch back and forth between the text screen and the graphics screen. This would be the case if the program contains INPUT statements and must also be used for drawing. This may appear to be inconvenient. On the other hand it assures that a drawing will not be disturbed by unwanted text.

fullscreen

is a procedure which causes the entire display screen to be filled by the graphics screen. The instruction would be used when working with turtle graphics to switch from the split screen (**splitscreen**) to the full graphics screen.

splitscreen

is a procedure which shows the graphics screen and a scrolling copy of the text screen with four lines of text and the cursor at the top of the display.

When used as a command, **USE turtle** does an automatic **splitscreen**, but not when it is used as a program statement.

clearscreen

is a procedure which deletes the entire graphics image no matter what the active (**viewport**) may be. To delete means to change all pixels to the background color.

clear

is a procedure which only deletes the graphics image within the drawing viewport.

Example:

viewport(0,100,0,100)	Only the 101 x 101 pixels
clear	in the lower, left-hand corner of display screen are cleared.

COLORS: In the following procedures with color specifications, the variable <color> must be an integer from -1 to 15. (Note: -1 means the background color.) See also Appendix B on colors and color codes.

---

textcolor(<color>)

is a procedure which defines the color of the characters on the text screen.

Example:

textcolor(0)	Black text is selected.

textbackground(<color>)

is a procedure which defines the background color of the text screen.

textborder(<color>)
is a procedure which defines the color of the text screen border.

pencolor(<color>)
is a procedure which defines the color of the pen.

Examples:

pencolor(7)	Yellow is selected as the drawing color.
pencolor(-1)	The background color is the drawing color.

background(<color>)
is a procedure which defines the graphics screen background color.

border(<color>)
is a procedure which defines the graphics screen border color.

getcolor(<x>,<y>)
is a function. Its value equals the color code of the pixel at location (<**x**>,<**y**>).

If (<**x**>,<**y**>) is outside the drawing area determined by the procedure **viewport**, then **getcolor(<x>,<y>)** returns the value -1.

The function **getcolor** does not change the current pen position.

Examples:
PRINT getcolor(1,2)
IF getcolor(0,0)<0 THEN move'center

plot(<x0>,<y0>)
is a procedure which places a dot at pen position **(<x0>,<y0>)**.

Example:
plot(4.3,56)

drawto(<x>,<y>)
is a procedure which draws a line from the current pen position to the point **(<x>,<y>)**, which becomes the new pen position.

Examples:
drawto(100,200)
drawto(-20,4000)

moveto(<x>,<y>)
is a procedure which moves the pen to the point (**<x>**,**<y>**).

Example:
moveto(200,-25)

draw(<dx>,<dy>)
is a procedure which draws a line from the current pen position (**<x0>**,**<y0>**) to the point with coordinates (**<x0>**+**<dx>**,**<y0>**+**<dy>**) and changes the pen position to the endpoint.

Examples:
draw(0,100)vertical line 101 units long
draw(-1.5,0.4)

move(<dx>,<dy>)
is a procedure which moves the pen without drawing from its current position (**<x0>**,**<y0>**) to the point with coordinates
(**<x0>** + **<dx>**, **<y0>** + **<dy>**).

Examples:
move(-3,20)
move(-2000,0)

setxy(<x>,<y>)
is a procedure which positions the pen at the point with coordinates (**<x>**,**<y>**). If the pen is down, this procedure draws a line just as **drawto(<x>,<y>)**. If the pen is up, it is moved just as with **moveto(<x>,<y>)**.

circle(<x0>,<y0>,<r>)
is a procedure which draws a circle with the center in (**<x0>**,**<y0>**) and radius **<r>**.

Whether the circle appears circular or elliptical depends upon your choice of the drawing region on the screen, the coordinate system and the adjustment of the vertical linearity of the TV or monitor screen. If the coordinate system has been selected in the drawing area so that the condition

$$\frac{\langle wxmax\rangle - \langle wxmin\rangle}{\langle wymax\rangle - \langle wymin\rangle} \star \frac{\langle vymax\rangle - \langle vymin\rangle}{\langle vxmax\rangle - \langle vxmin\rangle} = 1$$

on window and viewport bounderies is fulfilled, then the circle should appear to be perfectly round on the screen. If not, try adjusting the vertical linearity of the TV or monitor.

Example 1:
When **USE graphics** is called, it carries out the following procedures automatically:

```
viewport(0,319,0,199)
window(0,319,0,199)
```

The height/width ratio is equal to 1, and

circle(160,100,99)
will draw a round circle on the middle of the screen.

Example 2:

```
viewport(200,300,80,180)
window(-1,1,-1,1)
circle(0,0,1)
```

yields a round circle on the upper right-hand side of the screen.

arc(<x0>,<y0>,<r>,<a0>,<da>)

is a procedure which draws an arc with the center at (<x0>,<y0>) and radius of curvature <r>. The starting angle is <a0> degrees and the arc will subtend <da> degrees.

Examples:

```
arc(100,100,50,45,90)
arc(-20,25,30,15,-60)
```

xcor and *ycor*

are functions. They equal, respectively, the current x and y coordinates of the pen.

Examples:

```
PRINT xcor;ycor
plottext(xcor,ycor,"Figure 1")
```

fill(<x>,<y>)

is a procedure which uses pencolor to fill a region of the screen with color. The region to be filled in must contain the point (<x>,<y>). It must be bordered by a line or area of a *different* color or by an edge of the viewport.

fill does not alter the pen position.
See the summary example under the procedure **paint(<x>,<y>)**.

Example:
fill(10,56)

paint(<x>,<y>)

is a procedure which fills in a region of the screen with the drawing color. The region which is to be filled in must contain the point (<x>,<y>), and it must be bordered by a line or area with the *same* color or by an edge of the drawing area.

paint does not alter the current pen position.

Examples:
paint(-10,4)

pencolor(-1)	
paint(100,20)	A region is 'erased'.

The collection of examples below illustrates the differences between **fill** and **paint**:

```
USE graphics
graphicscreen(1)
pencolor(7)
drawto(319,199)
fill(10,100)          // if paint(10,100), no difference
pencolor(1)
circle(100,100,70)
fill(100,100)         // if paint(100,100), a difference!
WHILE KEY$=CHR$(0) DO NULL
```

showturtle

is a procedure which causes the turtle to be displayed on the graphics screen. The word 'turtle' is based on the use of relative graphics in the computer language Logo.

USE turtle	automatically causes the turtle to be shown.

hideturtle

is a procedure which causes the turtle on the graphics screen to become invisible.

turtlesize(<size>)

is a procedure which defines size of the drawing arrowhead (the turtle).

The parameter <size> must be a number between 0 and 10. When graphics is started up, this parameter is automatically set equal to 10.

home

is a procedure which places the turtle at coordinates (0,0) pointed upwards on the screen.

setheading(<heading>)

is a procedure which sets the direction in which the turtle points. If the turtle is visible, it will turn to face this direction.

<heading> is given in degrees:
0 corresponds to upwards.
90 is towards the right side of the screen.
-90 is towards the left.

USE turtle automatically sets the heading to 0.

heading

is a function which returns the value of the current heading. The heading is given in degrees with 0 towards the top of the screen, and 90 degrees towards the right.

penup

is a procedure which lifts the pen.

pendown

is a procedure which lowers the pen. It causes the turtle to draw as it moves.

When graphics is started up, the system automatically executes a **pendown**.

left(<angle>)

is a procedure which turns the turtle <**angle**> degrees to the left in relation to the current heading.

right(<angle>)

is a procedure which turns the turtle <**angle**> degrees to the right in relation to the current heading.

forward(<distance>)

is a procedure which moves the turtle <**distance**> units forward with the current heading. If the pen is down, a line is drawn.

back(<distance>)

is a procedure which moves the turtle <**distance**> units backwards in relation to the current heading. The turtle "backs up." If the pen is down, a line is drawn.

Summary example:
Press the <**f3**> function key (corresponding to the **USE turtle** command).

Write directly on the four text lines which are visible at the top of the screen:

```
left(90)
forward(70)
right(130)
forward(80)
left(40)
back(100)
hideturtle
```

The turtle has now drawn a number 4.

arcl(<r>,<da>)

is a procedure which draws a left-hand arc with a radius of curvature <**r**> and subtending an angle of <**da**> degrees. The starting point is the current turtle position, and the starting direction is the current heading.

Examples:
forward(20)
arcl(50,30)

After having drawn a straight line, the line curves towards the left, turning 30 degrees.

Procedure example (try it to see what happens):

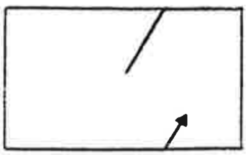
```
PROC soft'frame(xmin,ymin,width,height)
  IF width>20 AND height>20 THEN
    width=width-20
    height=height-20
    moveto(xmin+10,ymin)
    setheading(90)
    forward(width)
    arcl(10,90)
    forward(height)
    arcl(10,90)
    forward(width)
    arcl(10,90)
    forward(height)
    arcl(10,90)
  ENDIF
ENDPROC soft'frame
```

arcr(<r>,<da>)

is a procedure which draws a curve to the right with radius of curvature <**r**> and turning angle <**da**>. The starting point is the current position of the pen, and the initial heading is the current heading.

arcr(<r>,<da>) corresponds to **arcl(-<r>,-<da>)**.

Example:
arcr(3.45,50)



wrap

is a procedure which allows lines drawn on the graphics screen to continue beyond the edge of the screen, reappearing on the opposite side. For example, if the pen disappears at the top of the screen with x-coordinate 110 and heading 45, it will reappear at the bottom with the same x-coordinate and the same heading.

When **USE turtle** is engaged, the procedure **wrap** is carried out automatically. This however is NOT the case when **USE graphics** is started.

nowrap

is a procedure which terminates 'wraparound'. It can be restored with the procedure **wrap**.

textstyle(<width>,<height>,<heading>,<mode>)

is a procedure which is used to define how text printout will appear on the graphics screen. The actual printing of text is performed with the procedure **plottext**.

The parameters <width>, <height>, <heading> and <mode> must all be integers.

<width>	= letter width (1 corresponds to normal text.)
<height>	= letter height (1 corresponds to normal text.)
<heading>	= 0 , text is rotated 0 degrees (normal). 1 , text is rotated 90 degrees. 2 , text is rotated 180 degrees. 3 , text is rotated 270 degrees.
<mode>	= 0 , both the text and its background color is drawn. This means that the text area is cleared before new text is printed. 1 , only the characters of the text are printed. This means that a letter **a** placed on top of a letter **b** will not delete the entire letter **b**. Some of the remnants of the **b** will still be visible.

If a parameter is set equal to **-1**, then the current value is used.

On startup the computer automatically chooses **textstyle(1,1,0,0)**, corresponding to normal text size (as on the text screen) written horizontally, and both text and its background color is printed.

Example:

textstyle(2,1,2,0)	All subsequent text will be written upside down with characters of double width.
textstyle(3,2,-1,-1)	Only the text size is changed.

plottext(<x>,<y>,<text$>)

is a procedure which prints out the given text starting at the point (**<x>**,**<y>**).

The size of the letters, the orientation and writing mode are specified by the procedure **textstyle**.

plottext does not change the position of the pen.

Examples:

```
plottext(100,150,"COMAL")
text$:="What's my name?"
textstyle(1,3,1,0)
plottext(200,10,text$)
```

inq(<no>)

is a function which is used to obtain information concerning the state of the various graphics variables.

The parameter <**no**> must be an integer between 0 and 33.

<no>	Information	state	affected by
0	display	0 or 1	graphicscreen
1	text border	0 - 15	textcolors, border
2	text backgnd.	0 - 15	textcolors, textbackground
3	text color	0 - 15	textcolor, textcolors
4	graph. border	0 - 15	border
5	graph. backgnd.	0 - 15	background
6	pen color	0 - 15	pencolor
7	gr.text width	1 - 254	textstyle
8	gr.text height	1 - 254	textstyle
9	gr.text dirn.	0 - 3	textstyle
10	gr.text state	0 or 1	textstyle
11	turtle visible	TRUE,FALSE	showturtle, hideturtle
12	inside window	TRUE,FALSE	most drawing procedures
13	txt scrn seen	TRUE,FALSE	... screen
14	splt scrn seen	TRUE,FALSE	... screen
15	wraparound	TRUE,FALSE	wrap, nowrap
16	pen down	TRUE,FALSE	penup, pendown
17	x - position	integer	most drawing procedures
18	y - position	integer	most drawing procedures
19	vxmin	0-319	viewport
20	vxmax	0-319	viewport
21	vymin	0-199	viewport
22	vymax	0-199	viewport
23	wxmin	real number	window
24	wxmax	real number	window
25	wymin	real number	window
26	wymax	real number	window
27	COS(heading)	-1.0 - 1.0	seth,left,right,home,arcl,arcr
28	SIN(heading)	-1.0 - 1.0	seth,left,right,home,arcl,arcr
29	turtle size	0.0 - 10.0	turtlesize
30	x-aspect ratio	real number	=(wxmax-wxmin)/(vxmax-vxmin)
31	y-aspect ratio	real number	=(wymax-wymin)/(vymax-vymin)
32	x-text end	integer	plottext
33	y-text end	integer	plottext

savescreen(<filename$>)

is a procedure which saves a copy of the current graphics screen on diskette or tape. The file is saved under the name <**filename$**>.

The contents of the file are:

High resolution image (take up 36 blocks of 256 bytes):
0
background color
border color
1000 bytes for colors 0 and 1
8000 bytes for the bit pattern

Multi-color image (takes up 40 blocks of 256 bytes):
1
background color
border color
1000 bytes for colors 1 and 2
1000 bytes for color 3
8000 bytes for the bit pattern

Examples:

savescreen("gr0.drawing")	saves a high res image.
savescreen("gr1.circles")	saves a multi-color image.

loadscreen(<filename$>)

is a procedure which fetches an image which previously had been saved on diskette or on tape. See **savescreen**.

Examples:
loadscreen("gr0.drawing")
loadscreen("gr1.circles")

printscreen(<filename$>,<position>)

is a procedure which saves the contents of the current viewport to the file named <filename$>.

The parameter <position> is an integer from 0 to 479. It specifies the horizontal placement of the image on the MPS801 printer. Six <position> units correspond to one character from the edge of the paper.

The procedure is intended for getting a hard copy of a graphics image on the printer. But it can also be used, among other things, for saving a picture on diskette or on tape for later use.

Note that hard copy to a printer can only be done if the printer is compatible with the Commodore MPS 801.

High resolution graphics:

Printing Intensity	Color
0/4	background color
4/4	all other colors

Multi-color graphics:
Colors are printed according to a grey scale:

Printing intensity	**Color**
0/4	1: white
1/4	3: cyan, 7: yellow, 13: light green, 15: light grey
2/4	4: purple, 5: green, 8: orange, 10: pink, 12: grey, 14: light blue
3/4	2: red, 6: blue, 9: brown, 11: dark grey
4/4	0: black

Examples:

printscreen("lp:",79) — The graphics screen is dumped to a MPS 801 printer. The image begins right after the 13th character position.

printscreen("head",19) — The contents of the graphics screen are saved on diskette under the name **head**.

The file can not be fetched again using the procedure **loadscreen**, but must be entered instead as an ordinary sequential file. The following program segment fetches the saved file and prints it out on the printer:

```
OPEN FILE 2,"head",READ
SELECT OUTPUT "lp:"
WHILE NOT EOF(2) DO PRINT GET$(2,5000),
CLOSE FILE 2
SELECT OUTPUT "ds:"
```

Sprites

With your Commodore 64 it is possible to define a small graphics image which can be moved about on the graphics screen. Such an image is called a **sprite**.

Up to 8 sprites can be on the screen at one time. This makes it possible to create vivid graphics images with moving figures. Each sprite can be assigned its own color and be moved around independently of the others and the rest of the program. It is also possible to allow the sprites to interact with one another.

A number of procedures and functions are available for controlling sprites using the COMAL package **sprites**.

The package is made accessible by issuing the instruction:

USE sprites

You can imagine that you are working with sprites as follows:

You have a stage	(the display screen)
with a backdrop.	(the graphics background)
On the screen there are actors	(sprites)
which can move around	(using **movesprite**)
while performing an action.	(using **animate**)
The actors can move on and off the stage. The actors can move in front of and behind one another, and they can move in front of and behind the props	(graphics drawings)
You can direct the actors using sprite commands.	

A sprite shape is always created in a raster of 24 horizontal dots and 21 vertical dots, a total of 504 dots. A shape is defined by assigning a color to each dot. A high res shape has two colors, the background color and one more color. Because each dot corresponds to a bit in the computer memory you assign a foreground color to a dot by placing a **1** in the corresponding memory bit. All other dots are assigned the background color by a corresponding **0**.

Let's begin by making a sprite and moving it around the screen. This brief program shows how it can be done (it is called **Sprite 1** on the demo diskette/tape):

```
0100 DATA %00000000,%00000000,%00000000
0110 DATA %00000000,%00000000,%00000000
0120 DATA %00000000,%00000000,%00000000
0130 DATA %00001110,%00001110,%00000000
0140 DATA %00001111,%00011110,%00000000
0150 DATA %00000111,%00111100,%00000000
0160 DATA %00000011,%00110000,%00000000
0170 DATA %00000001,%11100000,%00000000
0180 DATA %00000011,%11100000,%00000000
0190 DATA %00000111,%11110000,%00000000
0200 DATA %00000011,%11100000,%00000000
0210 DATA %00110001,%11000000,%00000000
0220 DATA %00111111,%11100000,%00000000
0230 DATA %00001111,%11110000,%00000000
0235 DATA %00000111,%11110000,%00000000
0240 DATA %00000111,%11100000,%00000000
0250 DATA %00000111,%11100000,%00000000
0260 DATA %00011111,%11111000,%00000000
0270 DATA %00111110,%01111100,%00000000
0280 DATA %00000000,%00000000,%00000000
0300 DATA %00000000,%00000000,%00000000
0310
0320 USE graphics
0330 graphicscreen(0)
0340 USE sprites
0350 DIM drawing$ OF 64
0360 FOR i:=1 TO 63 DO
0370   READ byte
0380   drawing$:+CHR$(byte)
0390 ENDFOR i
0400 color:=1
0410 drawingno:=1
0420 spriteno:=1
0430 define(drawingno,drawing$+""0"")
0440 identify(spriteno,drawingno)
0450 spritecolor(spriteno,color)
0460 spritepos(spriteno,50,100)
0470 showsprite(spriteno)
0480
0490 WHILE KEY$=CHR$(0) DO NULL
0500
0510 movesprite(spriteno,250,150,200,0)
0520
0530 WHILE KEY$=CHR$(0) DO NULL
```

The DATA statements in lines 100-300 contain the definition of the figure.

These numbers (which can be written directly in binary in COMAL simply by prefixing binary numbers with the % sign) are read in the FOR-ENDFOR loop (360-390). The text string **drawing$** contains the bit pattern information which will form the sprite.

In line 430 this drawing is given the number 1. The extra ""**0**"" is included to specify that the drawing is a representation in high resolution graphics (as opposed to multi-color graphics).

In line 440 sprite 1 is identified to correspond to drawing no 1. In line 450 the color of the sprite with number 1 is specified (**color:=1,** i.e. white).

In line 460 sprite no 1 is placed on the screen so that the upper left hand corner of the figure is at (x,y) coordinates (50,100). Line 470 makes the sprite appear on the screen.

When you have had enough of the rabbit, press any key.

Line 510 causes the sprite to move over to the point with coordinates (250,150). The move is made in 200 steps. We will get back to the last 0 in the **movesprite** procedure call later.

When you again press any key, the program ends.

That was your first program using sprites. Now try giving the rabbit another color. Try moving it around to other points on the screen.

The Sprite is enlarged

Try adding the program line:

```
465 spritesize(spriteno,TRUE,TRUE)
```

Run the program again. The sprite has become twice as high and twice as wide!

More Sprites

Add the program lines

```
472 identify(2,drawingno)
474 spritecolor(2,0)
476 spritepos(2,80,100)
478 showsprite(2)
```

Try out the program. Can you make the new sprite move? See if you can make the two sprites start at either side of the screen. Make them move towards one another so that they exchange places.

You probably noticed that sprite no 1 passed in front of sprite no 2. The sprite with the lowest **spriteno** will always have first priority, so that the sprite with the lowest number will appear to pass in front of the other.

Two Sprites collide

The last number in the **movesprite** call determines how the sprite will move in relation to the other sprites and other graphics drawings on the screen. In the examples we have seen so far, it has been equal to 0.

If the number is changed to 1 in line 510, the sprite will be instructed to *detect a collision* with the other sprite. Both sprites will stop. Try it!

Saving a Drawing on Diskette

You can save a sprite shape using the instruction

saveshape(<drawingno>,<filename$>)

Drawings can be saved either on diskette or on cassette tape. (Notice: Use **cs:** in the file name to save on tape.) The drawing can be fetched for use in another program with the instruction

loadshape(<drawingno>,<filename$>)

This can obviate the need for including all the DATA statements in programs using the same sprite image.

The following program (**Sprite 2**) defines the drawing of the rabbit and saves this drawing on diskette under the name **sp0.rabbit**. If you run this program, you will e.g. be able to replace lines 100-310, 360-400 and 430 in other programs using the drawing with a single line:

430 loadshape(drawingno,"sp0.rabbit")

First the drawing must be saved using:
0100 to 0300: DATA statements with sprite image content (See previous program.)

```
0310
0320 USE sprites
0330 DIM drawing$ OF 64
0340 FOR i:=1 TO 63 DO
0350   READ byte
0360   drawing$:+CHR$(byte)
0370 ENDFOR i
0380 drawingno:=1
0390 define(drawingno,drawing$+""0"")
0400 saveshape(drawingno,"sp0.rabbit")
```

Sprites Used with Other Graphics

The following program **Sprite 3** shows how a sprite can be prepared to detect a collision with a graphics drawing and wait for the collision to happen. After the collision, the sprite can continue in a different direction.

0100 to 0300: DATA statements with sprite image content (See previous program.)

```
0310
0320 USE graphics
0330 graphicscreen(0)
0340 USE sprites
0350 color:=1
0360 DIM drawing$ OF 64
0370 FOR i:=1 TO 63 DO
0380   READ byte
0390   drawing$:+CHR$(byte)
0400 ENDFOR i
0410 drawingno:=1
0420 spriteno:=2
0430 define(drawingno,drawing$+""0"")
0440 identify(spriteno,drawingno)
0450 spritecolor(spriteno,color)
0460 spritepos(spriteno,50,100)
0470 showsprite(spriteno)
0480
0490 WHILE KEY$=CHR$(0) DO NULL
0500
0510 make'box
0520 movesprite(spriteno,250,150,200,4)
0530 WHILE NOT datacollision(spriteno,TRUE) DO NULL
0540 priority(spriteno,TRUE)
0550 movesprite(spriteno,130,180,50,0)
0560
0570 WHILE KEY$=CHR$(0) DO NULL
0580
0590 PROC make'box
0600   pencolor(8)
0610   moveto(100,10); draw(50,0)
0620   draw(0,150); draw(-50,0); draw(0,-150)
0630   fill(105,15)
0640 ENDPROC make'box
```

In line 520 the last number in the **movesprite** call is a 4. This causes the sprite to recognize collisions with graphics drawings. If 4 is changed to 0, the rabbit will move past the box without noticing it.

In line 530 there is a delay until a sprite-graphics collision occurs

In line 540 it is determined that the sprite will be seen behind the graphics drawing. Try changing TRUE to FALSE and re-run the program.

Sprite Cartoons

By switching two or more drawings quickly in succession, one can cause the rabbit to appear to perform actions while it moves.

We begin by making a few small changes in the drawing of the rabbit which we already have used. (This is easiest to do by listing the DATA statements and changing them directly.)

Next the order of the actions must be specified. This is done by means of the instruction **animate(<spriteno>,<action$>)**.

The completed program **(Sprite 4)** might appear as follows:

```
0100 DATA %00000000,%00000000,%00000000
0110 DATA %00000000,%00000000,%00000000
0120 DATA %00000000,%00000000,%00000000
0130 DATA %00001110,%00000000,%00000000
0140 DATA %00001111,%00011110,%00000000
0150 DATA %00000111,%00111111,%00000000
0160 DATA %00000011,%00110111,%00000000
0170 DATA %00000001,%11100000,%00000000
0180 DATA %00000011,%11100000,%00000000
0190 DATA %00000111,%11110000,%00000000
0200 DATA %00000011,%11100000,%00000000
0210 DATA %00000001,%11000000,%00000000
0220 DATA %00000011,%11100000,%00000000
0230 DATA %00111111,%11110000,%00000000
0240 DATA %00111111,%11110000,%00000000
0250 DATA %00000111,%11100000,%00000000
0260 DATA %00000111,%11100000,%00000000
0270 DATA %00011111,%11111000,%00000000
0280 DATA %00111110,%01111100,%00000000
0290 DATA %00000000,%00000000,%00000000
0300 DATA %00000000,%00000000,%00000000
0310
0320 DATA %00000000,%00000000,%00000000
0330 DATA %00000000,%00000000,%00000000
0340 DATA %00000000,%00000000,%00000000
0350 DATA %00001110,%00001110,%00000000
0360 DATA %00001111,%00011110,%00000000
0370 DATA %00000111,%00111100,%00000000
0380 DATA %00000011,%00110000,%00000000
0390 DATA %00000001,%11100000,%00000000
0400 DATA %00000011,%11100000,%00000000
0410 DATA %00000111,%11110000,%00000000
0420 DATA %00000011,%11100000,%00000000
0430 DATA %00110001,%11000000,%00000000
0440 DATA %00111111,%11100000,%00000000
0450 DATA %00001111,%11110000,%00000000
0460 DATA %00000111,%11110000,%00000000
0470 DATA %00000111,%11100000,%00000000
0480 DATA %00000111,%11100000,%00000000
0490 DATA %00011111,%11111000,%00000000
0500 DATA %00111110,%01111100,%00000000
0510 DATA %00000000,%00000000,%00000000
0520 DATA %00000000,%00000000,%00000000
0530
0540 USE graphics
0550 graphicscreen(1)
0560 USE sprites
0570 color:=1
```

```
0580 spriteno:=1
0590 DIM drawing$ OF 64, action$ OF 64
0600 FOR drawingno:=1 TO 2 DO
0610   drawing$:=""
0620   FOR i:=1 TO 63 DO
0630     READ byte
0640     drawing$:+CHR$(byte)
0650   ENDFOR i
0660   define(drawingno,drawing$+""0"")
0670 ENDFOR drawingno
0680
0690 identify(spriteno,1)
0700 spritecolor(spriteno,color)
0710 spritepos(spriteno,50,100)
0720 showsprite(spriteno)
0730 action$:=""1""+""4""+""2""+""5""
0740 animate(spriteno,action$)
0750 movesprite(spriteno,350,150,300,0)
0760
0770 WHILE KEY$=CHR$(0) DO NULL
```

We hope that this brief program example will inspire you to attempt your own complex dramatizations or games!

The order of the action is specified in line 730. Translating this line we find the following instructions: Display drawing 1 for 4 units of time, show drawing 2 for 5 units of time. Continue to repeat this action until the sprite stops.

See the overview under **animate** for further information on order of action sequences.

A Multi-Colored Sprite

So far we have only used drawings in high-resolution graphics (specified by a ""0"" in the **define(<drawingno>,<drawing$>=""0""))** procedure. The drawing is in only one color; it can readily be used either on a high-resolution graphics screen (**graphicscreen(0)**) or on a multi-color screen (**graphicscreen(1)**).

A sprite drawing can de created using several colors, but it is a little more complicated to create unless you can use the program **"Spriteeditor"** on the demo diskette or tape which accompanied your COMAL cartridge. See additional information on this program in Appendix H.

When a sprite image is defined using several colors, it is important to keep in mind that the horizontal neighboring pixels are associated in pairs when using multi-color graphics. In connection with the use of sprites in multi-color graphics, the following pairs of numbers determine the color of the sprite:

00 Transparent
01 Color 2
10 Foreground color 1
11 Color 3

Thus a sprite can be composed of 4 different colors, one of which is "transparent". The foreground color is determined by the **spritecolor** procedure. Colors 2 and 3 are determined by the **spriteback** procedure.

Just as with drawings in high-res graphics, it is a good idea to start by making a plan on graph paper. Pair the horizontal pixels when choosing the four possible "colors". Then prepare the drawing in the form of DATA statements as before. But now you must be more careful when assigning the correct number
combinations to the pixel pairs.

Here is a program (**Sprite 5**) which uses sprites with several colors:

```
0010 DATA %00000000,%00000000,%00000000
0020 DATA %00001010,%00000000,%00000000
0030 DATA %00001010,%00000000,%00000000
0040 DATA %00000101,%01010101,%01010000
0050 DATA %00000101,%01010101,%01010000
0060 DATA %00000101,%01010101,%01010000
0070 DATA %00001010,%10101010,%10100000
0080 DATA %00001010,%10101011,%11100000
0090 DATA %00001000,%00101011,%11100000
0100 DATA %00001000,%00101011,%11100000
0110 DATA %00001000,%00101011,%11100000
0120 DATA %00001000,%00101011,%11100000
0130 DATA %00001000,%00101001,%11100000
0140 DATA %00001010,%10101011,%11100000
0150 DATA %00001010,%10101011,%11100000
0160 DATA %00001010,%10101011,%11100000
0170 DATA %00001010,%10101011,%11100000
0180 DATA %00001010,%10101011,%11100000
0190 DATA %11111111,%11111101,%01111111
0200 DATA %11111111,%11111101,%01111111
0210 DATA %11111111,%11111101,%01111111
0220
0230 USE graphics
0240 graphicscreen(1)
0250 USE sprites
0260 DIM drawing$ OF 64
0270 FOR i:=1 TO 63 DO
0280   READ byte
0290   drawing$:+CHR$(byte)
0300 ENDFOR i
0310
0320 drawingno:=1
0330 define(drawingno,drawing$+""1"")
0340 background(0)
0350 spriteback(2,12)
0360 RANDOMIZE
0370 FOR spriteno:=0 TO 7 DO
0380   spritecolor(spriteno,RND(3,10))
0390   spritepos(spriteno,spriteno*40,50)
0400   identify(spriteno,drawingno)
0410   showsprite(spriteno)
0420   spritesize(spriteno,1,1)
```

```
0430 ENDFOR spriteno
0440 FOR i:=1 TO 100 DO plot(RND(0,319),RND(50,199))
0450 WHILE KEY$=CHR$(0) DO NULL
```

In line 240 multi-color graphics is selected. In line 330 the drawing is defined as a multi-color image by means of the ""1"" in the procedure call.

In line 340 the graphics screen background color is selected. In line 350 the 2nd and 3rd colors for the sprites are chosen.

In line 380 a random foreground color is chosen for each sprite. In line 420 all sprites are set double size. In line 440 stars are placed in the sky.

Sprite Overview

The package **sprites** contains 23 procedures and functions.

Definition of drawings and sprites:

define(<drawingno>,<drawing$>)
identify(<spriteno>,<drawingno>)

Sprite color(s):

spritecolor(<spriteno>,<color>)
spriteback(<color2>,<color3>)

Sprite size:

spritesize(<spriteno>,<xdouble>,<ydouble>)

Sprite position and motion:

spritepos(<spriteno>,<x>,<y>)
movesprite(<spriteno>,<x>,<y>,<dur>,<mode>)
startsprites
stopsprite(<spriteno>)
moving(<spriteno>)
spritex(<spriteno>)
spritey(<spriteno>)
animate(<spriteno>,<action$>)

Visibility:

showsprite(<spriteno>)
hidesprite(<spriteno>)
priority(<spriteno>,<graphics'in'front>)

Collision check:

spritecollision(<spriteno>,<yes/no>)
datacollision(<spriteno>,<yes/no>)

Information about sprites:b

spriteinq(<spriteno>,<property>)

A sprite is transformed into a graphics drawing:

stampsprite(<spriteno>)

Sprite images and storage:

saveshape(<drawingno>,<filename$>)

loadshape(<drawingno>,<filename$>)
linkshape(<drawingno>)
(Use **cs:** in file name for Datassette file.)

define(<drawingno>,<drawing$>)

is a procedure which defines a new drawing. The variable <**drawing$**> is a string with a length of 64 characters. It contains the information which specifies the sprite image. (See the examples at the beginning of this section.) The image defined is assigned the number given by the parameter <**drawingno**>.

There can be up to 32 images defined at one time. The parameter <**drawingno**> must be an integer between 0 and 31. The same image may be used to identify several different sprites.

Example:

define(23,house$)	The contents of the string **house$** defines drawing number **23**.

identify(<spriteno>,<drawingno>)

is a procedure which specifies that the sprite with the number <**spriteno**> is to be displayed using the image with the number <**drawingno**>. There can be up to 8 different sprites on the screen at one time. The parameter <**spriteno**> must be an integer from 0 to 7. The same drawing can form the basis for several sprites.

The sprite with the lowest <**spriteno**> has the highest priority and is therefore displayed in front of others with which it overlaps on the screen.

If the *graphics turtle* is displayed on the screen, it always has sprite number **7**.

Example:

identify(0,23)	Sprite number **0** is displayed as image no **23**.

spritecolor(<spriteno>,<color>)

is a procedure which assigns the sprite with the number <**spriteno**> the color specified. The parameter <**spriteno**> is an integer from 0 to 7, and <**color**> is an integer from 0 to 15. In high-resolution graphics the sprite will have this color. In multi-color graphics it is **color1**.

Example:

spritecolor(0,8)	Sprite number 0 is given color number 8.

spriteback(<color2>,<color3>)

is a procedure which specifies the colors in multi-color graphics. A multi-color sprite can have up to four colors:

transparent	(but does not cover other colors)
foreground color	set with **spritecolor** (=color1)
additional colors	set with **spriteback** (=color2 and color3)

Example:
spriteback(2,7) additional colors are red and yellow.

Special rules for Multi-colored Sprites:

In a multi-color drawing pixels are associated in horizontal pairs. Each color (background-, foreground- and additional) is indicated by bit patterns as follows:

Bit pair	**Color shown**	**Is set by**
00	transparent	graphics instructions
01	color 2	spriteback
10	color 1	spritecolor
11	color 3	spriteback

If graphics has priority over sprites (e.g. **priority(<spriteno>,TRUE)**), then **color2** with bit pattern **01** will also be the background color.

The parameter **color2** gives no report about collision with another sprite (**spritecollision**) or with graphics drawings (**datacollision**).

spritesize(<spriteno>,<xdouble>,<ydouble>)

is a procedure which determines whether the sprite numbered **<spriteno>** will be displayed in double size format. Normally a sprite occupies 24 pixels in the x-direction and 21 pixels in the y-direction. If **<xdouble>** is set equal to a number not equal to 0 (=**TRUE**), then the sprite will be shown in double width. Similarly for **<ydouble>**.

Examples:
spritesize(5,0,1) Sprite 5 double height
spritesize(2,TRUE,TRUE) Sprite 2 double size

spritepos(<spriteno>,<x>,<y>)

is a procedure which places the upper left-hand corner of the sprite at the point with screen coordinates (x,y).

Sprite positions are always specified in the screen coordinate system

independent of any other coordinate system which may have been defined by the graphics instruction **window**. Sprite coordinates are in fact specified in the coordinate system (-32768..32767, -32768..32767). Only the points (0..319,0..199) are visible on the screen.

Example:
spritepos(0,25,50) Sprite 0 is placed at screen position (25,50).

movesprite(<spriteno>,<x>,<y>,<dur>,<mode>)

is a procedure which moves the sprite numbered <**spriteno**> from the current position to the point (x,y). The motion is performed in <**dur**> small steps. Each step takes 1/50 of a second on computers using the European PAL standard. On computers using the American NTSC standard, each step takes 1/60 of a second. The time in each case corresponds to the time it takes to update the screen image.

The parameter <**dur**> expresses how many time intervals (screen updates) the movement will take. The fewer the number of steps, the faster the motion.

The parameter <**dur**> determines the speed of the sprite as follows:

1. If <**dur**> is held constant, then the speed will always be proportional to the distance between the two endpoints of the motion.
2. The speed will be independent of the distance between the endpoints if <**dur**> e.g. is defined by:

```
FUNC dur(spriteno,x,y)
  speed:=10
  dx:=x-spritex(spriteno)
  dy:=y-spritey(spriteno)
  dist:=SQR(dx*dx+dy*dy)
  RETURN dist*speed
ENDFUNC dur
```

   If this function is used to determine the parameter **dur**, the speed will always be constant. In this case about 1 screen time unit.
3. The speed can made independent of the **x-distance** (similarly for the **y-distance**), so that the sprite will appear to move with constant speed in one dimension.

   This can be assured if <**dur**> is determined by the following function.

```
FUNC dur(spriteno,x)
  speed:=10
  dist:=ABS(x-spritex(spriteno))
  RETURN dist*speed
ENDFUNC dur
```

If in particular **dur** equals **0**, the sprite will be moved immediately (next screen update) to the position (<**x**>,<**y**>) regardless of the value of <**mode**>. The sprite will not move again, but it can be caused to perform an action by using the procedure **animate**.

The parameter <**mode**> affects the moment when the movement begins, and determines whether or not collision with other sprites and graphics drawings will be taken into account. The parameter <**mode**> is an integer from 0 to 7:

<**mode**>	**effect**	
0	Start now,	no collision check
1	Await start signal,	no collision check
2	Start now,	check sprite/sprite collision
3	Await start signal,	check sprite/sprite collision
4	Start now,	check sprite/graphics collision
5	Await start signal,	check sprite/graphics collision
6	Start now,	check for any collision
7	Await start signal,	check for any collision

Note:
The procedure **movesprite** starts the motion. The COMAL system does not wait for the motion to stop but continues with the next line in the program. This makes it possible to start other sprites in motion, print messages, etc. Many things can be going on at the same time. If you do not want program execution to continue while the motion is carried out, you can add a **'wait'** line. For example:

WHILE moving(<spriteno>) DO NULL

or

WHILE NOT datacollision(<spriteno>,TRUE) DO NULL

Examples:

movesprite(2,200,130,100,0)	Move sprite no 2 to the point (200,130) in 100 screen updates. Start now with no collision check.
movesprite(0,250,-10,300,6)	Move sprite no 0 to the point (250,-10) in 300 steps. Start now, checking for sprite collisions and collisions with graphics drawings.

startsprites

is a procedure which initiates the motion of those sprites which are waiting for the start signal. See the **movesprite** procedure.

stopsprite(<spriteno>)

is a procedure which stops the motion of the sprite with the number specified.

moving(<spriteno>)

is a function which takes on the value TRUE (=1) if the sprite specified moves. Otherwise the value of function is FALSE (=0).

Example:

IF NOT moving(2) THEN movesprite(2,0,190,50,0)

If sprite 2 isn't moving, then it should be moved at once to screen coordinates (0,190).

spritex(<spriteno>) and ***spritey(<spriteno>)***

are functions which have the current x- and y-positions respectively as values.

Examples:

x'difference:=x-spritex(4)

y'difference:=y-spritey(4)

IF spritey(3)>200 THEN movesprite(3,spritex(3),20,200,0)

If sprite no 3 collides with the upper edge of the screen, then it 'falls' to the lower edge.

animate(<spriteno>,<action$>)

is a procedure which causes the sprite specified to automatically perform a given action. The action desired must be defined in the string **<action$>**.

The number of characters in the order of action specification must be an even number (maximum 64). Thus a maximum of 32 actions can be requested in each **<action$>** string.

Possible actions:

CHR$(<drawingno>)+CHR$(<time>)	the drawing with the number indicated should be displayed for the time specified.

Note that 0<= **<drawingno>** <=31 and 0<= **<time>** <=255 units of time (screen updates). See the procedure **movesprite** for more about timing.

If <**time**> is equal to **0**, the sprite will enter a wait state which can only be interrupted by the instruction **startsprites** or by a ”**g**”-action.

”p”+CHR$(<time>)	Pause for the given time interval.
”g”+CHR$(<spriteno>)	Restart the given sprite, if it is waiting.
”s”+CHR$(<spriteno>)	The specified sprite is shown.
”h”+CHR$(<spriteno>)	The specified sprite is hidden.
”x”+CHR$(<xdouble>)	If <**xdouble**> is **TRUE** (i.e. < > 0) the width of the sprite is doubled. If <**xdouble**> is **FALSE** (i.e. = 0), the sprite is 24 pixels wide.
”y”+CHR$(<ydouble>)	Analogous to **”x”+CHR$(<xdouble>)**.
”c”+CHR$(<color>)	The sprite acquires the color indicated, where 0<= <**color**> <=15.

The action must be started by the procedure **movesprite**. The actions specified by the string are carried out from left to right unless the sprite is in a wait state. When the last action has been completed, the sequence is repeated until the sprite is no longer in motion: either the **movesprite** motion is finished, or an **animate(<spriteno>,””)** instruction is executed.

Just as with the **movesprite** procedure the COMAL system does not wait for the action sequence to be completed but procedes directly to the next line in the program.

Note that **CHR$(<value>)** has the same meaning as ””<**value**>””, so

action$:=”s”+CHR$(1)+”p”+CHR$(10)+”h”+CHR$(1)+”p”+CHR$(10)
is identical to

action$:=”s”1”p”10”h”1”p”10””

Examples:

```
animate(1,”s”1”p”10”h”1”p”10””)
movesprite(1,100,100,0,0)
WHILE KEY$=CHR$(0) DO NULL
animate(1,””)
```

Sprite no 1 moves at once to the screen position (100,100) and flashes for 10 time units, until any key is pressed.

```
animate(3,””1””4””2””4””3””4””)
movesprite(3,300,180,500,0)
```

While sprite no 3 moves to screen position (300,180), it is first shown for 4 units of time as drawing no 1. Next it is displayed for 4 time units as drawing no 2, followed by drawing no 3. The sequence is then repeated again. Animation!

showsprite(<spriteno>)
is a procedure which makes the specified sprite visible (if it is on the screen).

hidesprite(<spriteno>)
is a procedure which conceals the sprite.

priority(<spriteno>,<graphics'in'front>)
is a procedure which determines the priority of the specified sprite in relation to the graphics drawings on the screen. If <**graphics'in'front**> has the value **TRUE** (=1), the graphics will be displayed in front of the sprite when they overlap. If the value is **FALSE** (=0), the sprite will appear in front of the graphics. When **USE sprites** is first used, the value is automatically set to **FALSE**.

Example:

priority(6,1)	Sprite no 6 will be displayed behind graphics.

spritecollision(<spriteno>,<yes'no>)
is a function which is used to specify when the given sprite collides with another sprite, or determine if it collided with one earlier.

If <**yes'no**>=TRUE, then **spritecollision** is FALSE, until a collision occurs.
If <**yes'no**>=FALSE, then **spritecollision** is TRUE, if a collision has already occured.

Collisions occur when colors different from the background color overlap. See in particular the remark under the **spriteback** procedure concerning multi-color graphics.

Examples:
WHILE NOT spritecollision(2,TRUE) DO NULL

Do nothing before sprite no 2 collides with another sprite.

IF spritecollision(4,0) THEN spritecolor(4,2)
If sprite no 4 has previously collided with another sprite, then it should be colored red.

datacollision(<spriteno>,<yes'no>)
is a function which is used to determine when the specified sprite collides with graphics drawings, or if it has previously collided with graphics drawings.

If <**yes'no**>=TRUE, then **datacollision** is FALSE until the collision occurs.

If <**yes'no**>=FALSE, then **datacollision** will be TRUE if a previous collision has occurred.

A collision takes place when colors different from the background color overlap. (See **spritecollision**.)

spriteinq(<spriteno>,<property>)

is a function which is used to obtain information concerning the sprite specified. The value of the parameter <**property**> determines which characteristic is to indicated.

<property>	The function	Range	Is set with
0	visible	TRUE/FALSE	hide/showsprite
1	Multi-color2 (01)	0..15	spriteback
2	Multi-color1 (10)	0..15	spritecolor
3	Multi-color3 (11)	0..15	spriteback
4	double width	TRUE/FALSE	spritesize
5	double height	TRUE/FALSE	spritesize
6	Multi-color	TRUE/FALSE	define,identify
7	graphics/sprite priority	TRUE/FALSE	priority
8	drawing number	0..31	identify
9	time remaining	0..215	movesprite
10	sprite/sprite collision	TRUE/FALSE	movesprite
11	sprite/graphics collision	TRUE/FALSE	movesprite
12	mode of motion	0..7	movesprite
13	number of actions	0..32	animate
14	no. of next action	0..32	animate

Note that **TRUE** and **FALSE** have the numerical values **1** and **0**.

Example:
FOR no:=1 TO 14 DO PRINT spriteinq(no)

stampsprite(<spriteno>)

is a procedure which is used to change the sprite into a graphics image. The sprite is "stamped" onto the graphics screen image.

Normally a sprite is not part of a graphics illustration and will therefore not be printed out with the rest of the graphics when the procedures **printscreen** and **savescreen** are used. The procedure **stampsprite** makes a copy of the sprite part of the graphics screen image. This procedure can be employed e.g. if you wish to incorporate the graphics turtle as part of a drawing which is to be saved or printed.

Example:
FOR spriteno:=7 TO 0 STEP -1 DO stampsprite(spriteno)
Copies of all visible sprites are made on the graphics screen.

saveshape(<drawingno>,<file name$>)

is a procedure which saves a copy of the sprite image on diskette or tape (remember **cs:** in the file name) under the name <**filename$**>. The drawing itself must be represented by a string 64 characters in length.

Example:
define(2,drawing$)
saveshape(2,"sp0.flower")

The figure contained in the string **drawing$**, is saved under the name "**sp0.flower**". The **0** is included in the name to indicate that the drawing is intended for use in high-resolution graphics.

loadshape(<drawingno>,<filename$>)

is a procedure which fetches a copy of the file named <**filename$**> from diskette or cassette tape. The file must have been saved previously using the procedure **saveshape**. The file <**filename$**> must contain a string with the definition of a sprite image. This drawing will be given the number <**drawingno**>.

Example:
loadshape(1,"sp0.flower") The file **sp0.flower** contains a string with an image which will be recognized as number **1** in the program.

linkshape(<drawingno>)

is a procedure which associates a copy of the drawing indicated with the COMAL program. When the program is saved using the instruction SAVE, the drawing will be saved with it. It can be read in later together with the program with the instruction LOAD.

If desired, the drawing can be disassociated from the COMAL program by using the instruction DISCARD.

The drawing must have been fetched earlier using the procedure **loadshape**. This drawing is assigned the number <**drawingno**>.

Example:
linkshape(7) The drawing with the number 7 is associated with the COMAL program in working memory.

Sound and Music

Those of you who are familiar with the sound capabilities of the Commodore 64 will be pleased to know that your COMAL cartridge offers you full and easy access to the Commodore 6581 **sound synthesizer (SID) chip**. This chip allows you to use up to three musical *voices* at the same time. In addition you have considerable freedom to decide how the individual notes will sound. You can control frequency, sound level, sound type, modulation and filtering. This section must be considered to be only an introduction to a very exciting subject. An entire book could be devoted to the study of music synthesis using the Commodore 64.

Using the COMAL instruction

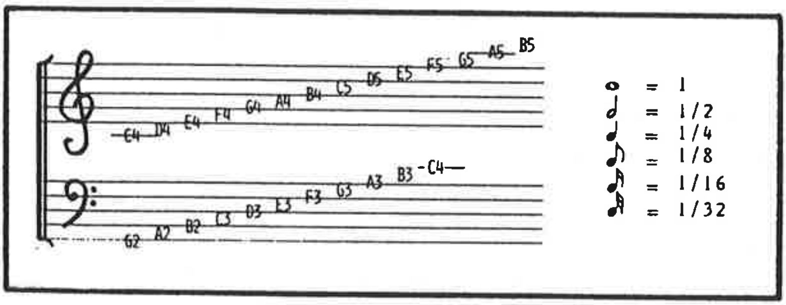
USE sound,

you make a number of additional procedures and functions available. Use these procedures and COMAL programming to create your own "orchestra".

Individual notes are denoted by strings. For example, "middle C" on the musical scale is denoted by the string variable **"c4"**.

The other notes in this octave are denoted: **"c4","c4#","d4","d4#"**, etc. Notes in the following octave are denoted by **"c5","c5#"** and so on. The notation for the preceding octave is **"c3","c3#",....** Notice that sharp notes are denoted "f4#" for "f-sharp" in the fourth octave, etc.

Although this tutorial is not intended to be a music course, here are a few facts which may be helpful when transfering a musical score to your Commodore 64. You will have to identify the notes and their durations. The following figure shows the ordering of some of the notes which can be played and the standard musical symbols for note duration:



The full range of notes starts with **"c0"** and extends up to and includes **"a7#"** on computers with European PAL standard, and **"b7"** on computer with the American NTSC standard.

In this section we will comment briefly on six programs, which you will find on the demonstration diskette or tape. You will find complete printouts of these programs in Appendix H. They are titled as follows and have the contents indicated below:

Music Demo: You will probably want to start by running this program to get an idea of the capabilities of your COMAL **sound package**. After examining the programs of lessons 1-5, you can return to study this program to see how all three voices can be used together.

Music 1: This program illustrates how individual notes are played.

Music 2: Up to three musical voices are available. It is possible to use up to three notes at the same time in your programs, giving your compositions a rich and realistic dimension.

Music 3: Here you can hear a demonstration or make your own composition using just one voice. Again, listing the program will be helpful to help you learn how to write your own music programs.

Music 4: This demonstration program allows you to change a number of parameters which affect the sound of each voice: **volume**, **soundtype** and the **adsr** (attack-decay-sustain-release) **waveform envelope**.

Music 5: Here is a complete composition illustrating synchronized music with several voices.

After trying out the **Music Demo**, you will probably want to LOAD, RUN and LIST each of the five "Music" programs. Notice that the instruction **USE sound** must appear in a program, before the sound control instructions will be active. In the listing for **Music 1** pay particular attention to lines 150-320:

```
0150 INPUT AT 8,1: "voice: ": voice
0160 INPUT AT 9,1: "note-code: ":code$
0170 play(voice,code$)
0190
0200 PROC play(voice,code$)
0210   IF code$<>"z" THEN
0220     note(voice,code$)
0230     gate(voice,1) // attack & decay
0240   ENDIF
0250   delay(16) // sustain
0260   gate(voice,0) // release
0270 ENDPROC play
0280
0290 PROC delay(sec'32)
0300   TIME 0
0310   WHILE TIME<1.875*sec'32 DO NULL
0320 ENDPROC delay
```

The first thing that happens are the INPUT statements. The **voice number** (1, 2 or 3) and the **note code** (c0,c0#,... or a7#) are to be entered here. In line 170 the **sound** procedure **play** is called with these two variables as inputs.

If the note code variable **code$** is a "**z**", no new note will be played. Use "**z**" when you want a pause to occur in your music. It must be followed by a duration code, just like a note. If **code$** is a legal note code, then the note will be played.

This is accomplished as follows. The procedure **note(voice,code$)** sets up the voice and the note, getting it ready to be played. The procedure **gate(voice,1)** initiates the playing of the note; the attack and decay portions of the *adsr envelope* are executed at once. The procedure **delay** (which must be provided by the user) determines the length of time the note is sustained. Finally, the instruction **gate(voice,0)** terminates the sustain phase and the note procedes to decay, as specified by the **adsr** procedure. More on **adsr** later!

The user supplied **delay** procedure can be any routine which can use up a well-defined time interval. In this program we have done this by means of a WHILE...DO loop which does nothing (NULL). The procedure call **delay(16)** in line 250 causes a delay of 16/32 = 1/2 second.

To make two notes play simultaneously, instructions like the following must be added:

```
225 note(2,"c5")
235 gate(2,1)
265 gate(2,0)
```

Try LOADing, RUNning and LISTing **Music 2**. You will find the procedures **play** and **delay** used again. In addition you will find the following instructions:

```
0130 FOR voice:=1 TO 3 DO
0140   soundtype(voice,3)
0150 ENDFOR voice
0160
0170 INPUT AT 7,1: "note-code: ": code$
0180
0190 FOR voice:=1 TO 3 DO
0200   PRINT AT 10,1: "voice ";voice
0210   play(voice,code$)
0220   play(voice,"z")
0230 ENDFOR voice
```

Lines 130-150 are used to set up the **soundtype** of each of the three voices. This is a **sound** package instruction with two input variables. The first variable is the **voice** number (1,2 or 3), and the second one is the **soundtype** (0,1,2,3 or 4). These numbers specify **soundtypes** as follows:

soundtype 0: silence
soundtype 1: triangular wave
soundtype 2: sawtooth wave
soundtype 3: square wave
soundtype 4: white noise

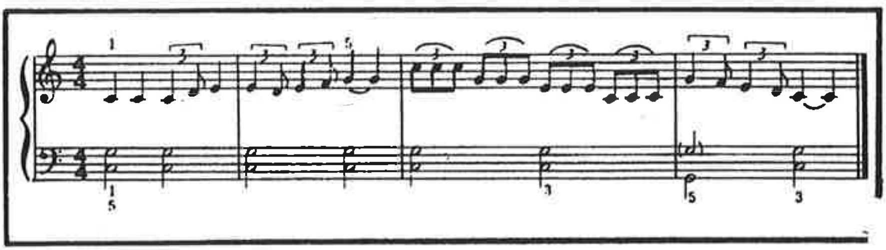
It will require some experience before you become skillful at selecting the best soundtype to achieve the effects you want. Lines 130-150 in this example set all three voices to the **square wave** soundtype.

Line 170 inputs a note code. Lines 190-230 allow the note to be played using all three voices, so that you can experience the differences among them. Notice that the procedure **play(voice,code$)** is used just as it was used earlier.

Notice also that we have used "**z**" as an input to **play** to achieve a pause between the playing of each note. Try removing line 220 and listen to what happens when the program is run.

Now LOAD and RUN the program **Music 3**. LIST it, and pay particular attention to lines 330-500:

```
0330 PROC play'melody // Row, Row, Row Your Boat
0340
0350 melody:
0360    DATA "c4",8,"z",2,"c4",8,"z",2,"c4",8,"d4",4
0370    DATA "e4",8,"z",8,"e4",8,"d4",4,"e4",8
0380    DATA "f4",4,"g4",16,"z",8,"c5",4
0390    DATA "c5",4,"c5",4,"g4",4,"g4",4
0400    DATA "g4",4,"e4",4,"e4",4,"e4",4
0410    DATA "c4",4,"c4",4,"c4",4,"z",8,"g4",8
0420    DATA "f4",4,"e4",8,"d4",4,"c4",8
0430
0440 RESTORE melody
0450 WHILE NOT EOD DO
0460    READ code$,sek'32
0470    play(voice,code$)
0480 ENDWHILE
0490
0500 ENDPROC play'melody
```

This procedure plays a simple tune (**Row, Row, Row your Boat**):

The procedure starts by zeroing the DATA pointer (**RESTORE melody**), so the DATA statements are read from the beginning each time the melody is played. The lines of DATA contain pairs of information (note codes and their durations). Lets take a quick look at the data to see how it relates to the simple piece of music in this illustration.

Look at the music. The first note is "middle C" with the note code **c4**. It is a quarter note. If we decide to give a whole note a duration of **32**, then the quarter note must be given a duration of **8**. The first two data elements are **"c4",8**. Notice that the first element is a **string variable**, while the second element is an **integer**. After the first note we want a brief pause, so the notes don't all run together. We enter **"z",2** to accomplish this. The next two notes are also middle C, so they are entered in the same way. The vertical line in the musical score indicates a brief pause, so we have entered a **"z",8** for this purpose. Notice that it is not always necessary to enter a pause between notes. You must experiment until you understand how to achieve the effect you want.

There are many ways of handling the music data. You could enter lines of music as long strings of data and design a procedure to "pick out" the note codes and delays one at a time. You might choose to make the duration codes integer variables to save memory when composing a lengthy piece. If sections of the music are repeated, then it will be a distinct advantage for you to design each unique section of the music as an independent procedure. A "master procedure" can then be written to play the piece, executing each section in turn.

The actual playing of the notes is accomplished in lines 450-480. Data is entered a pair at a time (note code and duration). The note is played by **play(voice,code$)**. And this process continues until there is no more data (EOD is TRUE).

Turn now to **Music 4**. This program will help you to experiment with a few more instructions from the sound package. The following lines are of particular interest:

```
0180 INPUT AT 11,1: "VOICE (1/2/3)? ": voice
0190 INPUT AT 13,1: "VOLUME (0-15)? ": vol
0200 INPUT AT 15,1: "SOUNDTYPE (1/2/3/4)? ": type
0210 soundtype(voice,type)
0220 volume(vol)

0420 INPUT AT 21,1: "A,D,S,R? ": a,d,s,r
0430 adsr(voice,a,d,s,r)
0440
0450 play'melody
```

Lines 180-200 input the **voice number**, **music volume** and the **soundtype** for the voice selected. The package procedure **volume(vol)** can be used to regulate the volume from silence (0) to the maximum value (15).

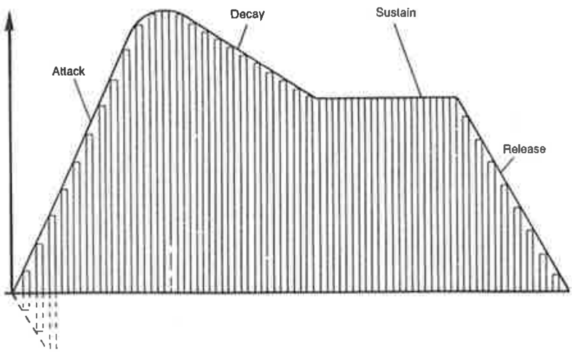
In line 430 the user can select the **waveform parameters**. These determine the shape of the sound intensity pattern which forms the note. The actual sound consists of waves as specified by the **soundtype** procedure. The **adsr** procedure allows the user to control the shape of the "envelope" governing how the note rises in intensity (*attack*), *decays*, is *sustained* at a certain level then dies away (*release*). Notice that the duration of the sustain phase of the note is regulated by means of the user procedure **delay**. The shape of the envelope is specificed by the following numbers, each of which can be chosen freely in the range from 0-15:

Attack specifies the rate at which the waveform envelope rises. This rate should be high (i.e. the attack parameter small) to achieve a "piano", "banjo" or "harpsicord" sound. The sound of plucked stringed instruments is characterised by a very audible attack phase when the note is struck.

Decay determines how fast the note dies down to the sustain level. Varying this number will vary the type of stringed instrument, you want to emulate.

Sustain defines the intensity level at which the note will be played for the delay period specified by the user's **delay** procedure.

Release regulates how fast the note "dies away" at the end of the sustain period.



The last program, **Music 5**, illustrates how several voices can be played at once using the procedure **playscore**. In this example only one voice is used (voice 1). We will see later how this can be changed by adding a few more lines.

The notes should first be read in and transformed to frequency values by means of the function **frequency**. All these numbers are then stored in a table of integers **tone#()** along with the associated duration data: an

ads'pause for the attack-decay-sustain phase and an **r'pause** for the release phase (including the delay between notes). The numbers are brought into the voice 1 register by means of the procedure **setscore**. Then the playing is initiated by the procedure **playscore**.

While the melody is played, the following COMAL program prints out some numbers. This is done here simply to illustrate that while the SID chip is at work playing music, the processor can proceed with other tasks. When the background music is finished, the function **waitscore** takes on the value TRUE (=1). Thus the printing of numbers in the WHILE-ENDWHILE loop will stop when the music stops.

```
0090 no:=0
0100 WHILE NOT EOD DO
0110   no:+1
0120   READ code$,tim
0130   tone#(no):=frequency(code$)
0140   ads'pause#(no):=tim*2
0150   r'pause#(no):=tim*2
0160 ENDWHILE
0170
0180 tone#(nr+1):=0
0190 setscore(1,tone#(),ads'pause#(),r'pause#())
0200 playscore(1,0,0)
0210
0220 number:=0
0230 WHILE NOT waitscore(1,0,0) DO
0240   number:+1
0250   PRINT number;
0260 ENDWHILE
```

Add the lines:

```
192 setscore(2,tone#(),ads'pause#(),r'pause())
194 setscore(3,tone#(),ads'pause#(),r'pause#())
```

and change lines 200 and 230 to:

```
0200 playscore(1,1,1)
0230 waitscore(1,1,1)
```

The three voices will play the melody simultaneously (synchronized). The program ends, when all three voices have finished.

Can you write a "round" with a delay between the different voices?

Notice that when the package is first brought into play with the instruction **USE sound**, the following default values are selected:

```
adsr(1,0,4,12,10)
adsr(2,10,8,10,9)
adsr(3,0,9,0,9)
FOR voice:=1 TO 3 DO
  pulse(voice,2048)
  setfrequency(voice,0)
ENDFOR voice
volume(15)
soundtype(1,1) // piano
soundtype(2,2) // violin
soundtype(3,3) // cymbal
```

The intention of the five introductory music programs has been to acquaint you with how to control the sounds created by the **sound** package. At first you may feel that there is a great deal to learn before you can compose music. This is true. But as with many other situations, a skill worth learning does take time and effort. Be patient, experiment and be curious. As you solve each problem which arises, you will learn something new!

We conclude this section with a summary of the instructions made available when you invoke the **sound** package:

```
volume(<level>)
note(<voice>,<code$>)
gate(<voice>,<start'stop>)
soundtype(<voice>,<soundtype>)
adsr(<voice>,<attack>,<decay>,<sustain>,<release>)
setscore(<voice>,<frequency()>,<pause1()>,<pause2()>
playscore(<voice1>,<voice2>,<voice3>)
stopplay(<voice1>,<voice2>,<voice3>)
waitscore(<voice1>,<voice2>,<voice3>)
frequency(<code$>)
setfrequency(<voice>,<frequency'value>)
sync(<voice'combination>,<yes'no>)
filterfreq(<frequency'value>)
filter(<voice1>,<voice2>,<voice3>,<external>)
filtertype(<low>,<band>,<high>,<3-interrupt>)
pulse(<voice>,<pulse'width>)
ringmod(<voice'combination>,<yes'no>)
resonance(<degree>)
env3
osc3
```

Sound Instructions in Depth

volume(<level>)

is a procedure which controls the common sound level for all three voices. The parameter <**level**> is an integer from 0 to 15.

Example:

volume(15)	maximum sound level

note(<voice>,<code$>)

is a procedure which is used to indicate the tone <**code$**> which the voice with the number <**voice**> will play. The parameter <**voice**> can be 1, 2 or 3; <**code$**> is a string with possible values: "c0", "c0#","d0",...,"a7#" on machines using the European PAL standard. On machines using the American NTSC standard tones up to "b7" can be played. The letters in each note code indicate the note, and the number indicates the octave. The character # indicates half notes (sharp notes).

Example:

note(2,"d5")	voice **2** will play the note **d5**

gate(<voice>,<start'stop>)

is a procedure which either starts or stops the playing of voice number <**voice**>. If the parameter <**start'stop**> equals **1**, the note starts. If <**start'stop**> equals **0**, it stops.

Example:

gate(3,1)	Voice **3** starts playing.

soundtype(<voice>,<soundtype>)

is a procedure which is used to indicate which <**soundtype**> <**voice**> is to be. The parameter <**soundtype**> is the periodic base signal which will be used to create the notes. It can be any of the following:

<**soundtype**>
0: silence
1: triangle waveform
2: sawtooth wave
3: square wave
4: white noise

Example:

soundtype(1,3) voice **1** formed with square waves

adsr(<voice>,<attack>,<decay>,<sustain>,<release>)

is a procedure which determines the shape of the waveform envelope. See the program, **Music 4**. Note especially that <**sustain**> indicates a sound level from 0 to the maximum sound **level** (determined by **volume**), while <**attack**>, <**decay**> **and** <**release**> control the **time** dependence.

Value	<attack>	<decay> and <release>:
0:	2 msec	6 msec
1:	8	24
2:	16	48
3:	24	72
4:	38	114
5:	56	168
6:	68	204
7:	80	240
8:	100	300
9:	250	750
10:	500	1.5 sec
11:	800	2.4
12:	1 sec	3
13:	3	9
14:	5	15
15:	8	24

<**sustain**> be equal to 0, 1,..., 15

Example:

adsr(1,13,13,8,13) voice **1** envelope is specified

setscore(<voice>,<frequency()>,<pause1()>,<pause2()>

is a procedure which is used to store a melody in the register of the given voice. <**voice**> must be an integer 1, 2 or 3. <**frequency()**> is an array which is to contain the frequency of the individual tones, the last value being 0 to turn off the voice. The corresponding delays are stored in the arrays <**pause1()**> (the ads-delays) and in <**pause2()**> (the r-fase delays).

The function **frequency** can be used for translating notes to these frequency numbers. The playing of the tune itself is initiated by the procedure **playscore** and stopped by the procedure **stopplay**.

Example:
setscore(2,freq(),ads'pause(),r'pause())
Frequencies with corresponding pauses are stored in the register of voice no 2.

playscore(<voice1>,<voice2>,<voice3>)

is a procedure which is used to synchronize the start of the voices. A **1** in the variable position corresponding to <**voiceX**> starts the voice playing:

Example:
playscore(1,1,0) voice **1** and **2** are started

stopplay(<voice1>,<voice2>,<voice3>)

is a procedure which stops the playing of the voices indicated. If <**voiceX**> is TRUE (=1), then voice X stops playing.

Example:
stopplay(0,1,1) voice **2** and **3** are stopped

waitscore(<voice1>,<voice2>,<voice3>)

is a function which returns the value TRUE (=1) if the playing of the indicated voice combination has finished.

Example:
WHILE NOT waltscore(1,1,0) DO NULL
do nothing before voice **1** and **2** have finished playing.

frequency(<code$>)

is a function which returns the integer value which the SID chip must receive to play the note. It is mostly used to compute array values for the procedure **setscore**. The integer value lies between -32768 and 32767 inclusive. It is NOT possble to transform notes between octaves directly by dividing these numbers by 2. The parameter <**code$**> must contain a string with a valid note code (i.e. one of the codes "c0", etc.).

Example:
frequency("c4") the note "**c4**" is transformed to a number

setfrequency(<voice>,<frequency'value>)

is a procedure which is used to define the frequency of each <**voice**>.

The number <**frequency'value**> must be in the range 0 - 65535. These numbers do not correspond directly to the SID chip frequency codes.

Example:
setfrequency(2,2000)

sync(<voice'combination>,<yes'no>)

is a procedure which takes care of synchronization with respect to the <**voice'combination**> indicated if <**yes'no**> **equals 1.** Otherwise the voice combination is not synchronized.

Note:	**voice'combination number:**	**corresponds to sync between voices:**
	1	1 and 3
	2	1 and 2
	3	2 and 3

Example:

sync(1,1) **voice 1** and **3** are syncronized

filterfreq(<frequency'value>)

is a procedure which is used to determine the cutoff frequency for the filter. The parameter <**frequency'value**> must be in the range 0 to 2047 inclusive, corresponding to frequencies between about 30 and 12000 Hz.

Example:
filterfreq(729) Middle C

filter(<voice1>,<voice2>,<voice3>,<external>)

is a procedure which is used to select which voices are to be filtered, i.e. damped. A **1** in a <**voiceX**> position means that voice X is to be filtered.

Example:
filter(0,1,1,1) voice **1** should NOT be filtered

filtertype(<low>,<band>,<high>,<3'interrupt>)

is a procedure which is used to select the filter type.

If <**low**> equals **1**, then a 'low-pass' filter is used, damping tones in the treble range. All frequencies above the filter frequency (set by **filterfreq**) are damped 12 dB per octave.

If <**band**> equals **1**, then damping occurs on both sides of the filter frequency; 6 dB per octave.

If <**high**> equals **1**, then the low frequencies are damped by 12 dB per octave.

If <**3'interrupt**> equals 1, then voice 3 will not be audible. It can be used to code information about synchronization and ringmodulation.

Several filters can be selected at the same time.

Example:

filtertype(1,0,1,0)	creates a "notch filter" which has the opposite effect of a "band-pass filter": damping occurs around the filter frequency.

pulse(<voice>,<pulse'width>)

is a procedure which is used to indicate the ratio between the time during which a square wave is high and the time during which it is low (the "duty-cycle"). The more this ratio deviates from 1:1, the more "nasal" and "sharp" the sound will be. The parameter <**pulse'width**> is a number from 0 to 4095 inclusive. When selected as 2048 the ratio is 1:1.

Example:

pulse(1,2048)	The ratio high/low equals 1.

ringmod(<voice'combination>,<yes'no>)

is a procedure which is used to determine whether ring modulation is to be in effect. The parameter <**voice'combination**> selects which voices are affected (see **sync**). If <**yes'no**> is TRUE (=1) modulation will occur; it will not if <**yes'no**> equals FALSE (=0).

When ring modulation is in effect, then two new voices with frequencies equal to the sum and the difference between the original voices are generated.

resonance(<degree>)

is a procedure which is used to indicate to what degree certain frequencies will be emphasized. The greater the value of the parameter **degree**, the greater the emphasis on the frequencies selected by the procedure **setfrequency** will be. This will give the sound a synthetic quality. The parameter <**degree**> must be an integer from 0 to 15.

env3

is a function with no parameters. It returns the amplitude of the intensity envelope for voice number 3. The values of the function lies in the interval 0 - 255.

Displaying the intensity envelope:

```
USE sound
USE graphics
graphicscreen(0)
volume(10)
soundtype(3,1)
note(3,"a4")
adsr(3,13,13,8,13)
gate(3,0)
WHILE env3<>0 DO NULL
TIME 0
gate(3,1)
WHILE TIME<60*10 DO
  drawto(TIME/5,env3/256*199)
ENDWHILE
gate(3,0)
WHILE TIME/5<320 DO
  drawto(TIME/5,env3/256*199)
ENDWHILE
WHILE KEY$=CHR$(0) DO NULL
```

osc3

is a function with no parameters. It returns a value from 0 to 255. The number indicates the excursion of the current sound type of voice 3. In the case of a **triangle** the numbers vary from 0 to 255 and back to 0 again. For the **sawtooth** wave, values increase from 0 to 255 then fall rapidly back to 0. The **square wave** pulse varies between 0 and 255. **White noise** yields random numbers from 0 to 255.

Note that the sound continues playing after a COMAL program stops. The sound stops only if a melody is finished, if the COMAL program produces an error message or if it communicates with the disk drive. These instructions all use the *interrupt*, also used by the sound chip.

Packages for using the Control Ports

The COMAL cartridge contains 3 packages which can be used with the two input ports (game ports) on the right hand side of your Commodore 64 (on the back of the SX-64). These two inputs will be refered to as *control port 1* and *control port 2*.

The control ports can be used to attach accessories like joysticks or paddles. Signals from these devices can be interpreted and assigned numbers by the the computer. The Commodore 64 can be used with a range of different accessories - both commercially available and those you can build yourself. (See Chapter 7 on Peripheral Equipment.)

In this section we will deal specifically with:

paddles
joystick
light pen

These accessories can be purchased from your Commodore dealer.

Some of the COMAL packages contain procedures which make it easier to use these accessories.

Paddles

The package **paddles** is made available by the instruction:

USE paddles

A pair of paddles should be attached to a control port. The paddles will be refered to as **paddle a** and **paddle b**. Each paddle has a *knob*, which is used to change the position of a variable resistor, and a *push-button*, which shorts a port input to ground when activated.

The package contains a single procedure:

paddle(<portno>,<a'paddle>,<b'paddle>,<a'button>,<b'button>)

which transforms information from the control port to numbers.

* The parameter <**portno**> must contain the number of the control port to which the paddle pair is attached: **1** or **2**.
* The variables <**a'paddle**> and <**b'paddle**> contain the numerical value corresponding to the knob position of paddle a and paddle b respectively:

  0<= <**a'paddle**> <=255 and 0 <= <**b'paddle**> <=255

* The variable <**a'button**> equals **1** if the a-pushbutton is depressed; otherwise <**a'button**> equals **0**. Similarly for <**b'button**>.

Example:

```
USE paddles
paddle(2,a'paddle,b'paddle,a'button,b'button)
PRINT a'paddle;b'paddle;a'button;b'button
```

The signal values are fetched from control port 2 and printed out in the next line.

The following program example, **Paddle Game**, is available on the demo diskette (tape):

```
0010 USE paddles
0020
0030 DIM format$ OF 40
0040 format$:=" ###  #  ###   #"
0050
0060 PAGE
0070 INPUT AT 2,1: "control port no > ": portno
0080
0090 DIM winner$ OF 1
0100 winner$:="c"
0110 PRINT AT 9,2: "Who can adjust the paddle and press "
0120 PRINT AT 10,2: "the fire button the fastest?"
0130 PRINT AT 13,2: "Press a key to start."
0140 RANDOMIZE
0150 WHILE KEY$=CHR$(0) DO NULL
0160 number:=RND(0,255)
0170 PRINT AT 15,2: "The number is: ",number
0180
```

```
0190 REPEAT
0200   paddle(portno,a'paddle,b'paddle,a'button,b'button)
0210   PRINT AT 5,1: " a'paddle  a'button   b'paddle  b'button"
0220   PRINT AT 6,1: USING format$: a'paddle,a'button, b'paddle,b'button
0230
0240   IF number=a'paddle AND a'button THEN winner$:="a"
0250   IF number=b'paddle AND b'button THEN winner$:="b"
0260 UNTIL winner$ IN "ab"
0270
0280 PRINT AT 17,2: winner$+" was fastest!"
```

Joysticks

The package **joysticks** becomes accessible when you use the instruction:

USE joysticks

Attach a joystick to one of the control ports. A *joystick* is a peripheral device which can be centered or moved by the user into any of 8 different positions:

Direction	*COMAL - number*
up	1
up-left up-right	8 2
left neutral right	7 0 3
down-left down-right	6 4
down	5

In addition there is a push-button on the joystick (the fire button) which sends a signal to the computer when pressed.
The package contains a single procedure:

joystick(<portno>,<direction>,<button>)

which translates the signals from the joystick to numerical values for use in programs.

* <**portno**> must contain the number of the port to which the joystick is attached: **1** or **2**.
* <**direction**> is a variable which equals a number in the range **0 - 8**. These values indicate the position of the joystick. See above.
* <**button**> is a variable with the value **1** when the fire button is pushed, otherwise <**button**> equals **0**.

Example:

```
USE joysticks
joystick(2,direction,button)
PRINT direction;button
```

The signal values are fetched form control port 2 and printed in the next line.

The program example shows how a joystick can be used to draw:

```
0100 PAGE
0110 PRINT "JOYSTICK FOR DRAWING"
0120 PRINT
0130 PRINT "The joystick determines drawing direction."
```

```
0140 PRINT "The fire button switches colors."
0150 PRINT
0160 PRINT "Press <STOP> to stop the program."
0170 PRINT "Press <f5> to see the drawing again,"
0180 PRINT "and <f1> to get back to the text."
0190 PRINT
0200 INPUT "Joystick in port no: (1 or 2) :": portno
0210 IF portno<1 OR portno>2 THEN portno:=2
0220
0230 USE turtle
0240 USE joysticks
0250 graphicscreen(1)
0260 background(1)
0270 pencolor(5)
0280
0290 LOOP
0300   joystick(portno,direction,button)
0310   IF direction THEN
0320     setheading((direction-1)*45)
0330     forward(1)
0340   ENDIF
0350   IF button THEN // change color
0360     pencolor((inq(6)+1) MOD 16)
0370   ENDIF
0380 ENDLOOP
```

Light Pen

In order to understand how a light pen works, you have to know something about how the picture on your TV or monitor screen is formed. The picture is created by an electron beam which scans back and forth across the face of the screen at high speed. As it scans, the intensity of the beam changes. Phosphors on the inside surface of the screen react to the electron beam by emitting light, thus creating a visible image. The picture on the screen is updated 50 or 60 times each second, so the eye doesn't notice this process. A light pen contains a photodiode in its tip. It can detect variations in the light level striking it.

When the electron beam passes the point on the screen where the light pen is positioned, it can be illuminated. If it is illuminated and a signal is sent to the computer, the instant when the signal arrives corresponds to a particular position on the screen.

The light pen should always be connected to control port 1. Next make the package **lightpen** accessible with the instruction:

```
USE lightpen
```

The light pen works best when the screen border is dark and the background is light.

If the program segment listed below does not work right away, then try adjusting the contrast and brightness adjustments on your display.

Using this program you can experiment with the operation of the light pen. Type in the program and try it to find the offset:

```
0010 PAGE
0020 USE lightpen
0030 USE system
0040 textcolors(0,14,6)
0050
0060 offset(0,0)
0070 REPEAT readpen(x,y,ok) UNTIL ok
0080
0090 PRINT x;y
```

The program contains 2 procedures from the light pen package: Line 60 specifies that the light pen's measurement of the coordinates of a point should not yet be offset. Line 70 detects where on the screen the light pen is pointed.

Move the pen slowly from the dark edge in the lower left hand corner into the light area. The program will then print out the light pen's measurement of the coordinates of this point. Try a few times until the coordinates have been determined with reasonable accuracy. These coordinates are referred to as the light pen's **offset** from (0,0). We will term this coordinate pair (<xoff>,<yoff>). The coordinates (<xoff>,

<yoff>) can vary from display to display due to delays in the electonic detection process.

In line 60 of the program the **offset** was set equal to (0,0). Now change this to the values of (<xoff>,<yoff>) which you have just found.

When you run the altered program and move the light pen in and out of the corner, it should now register the coordinates (0,0). If it does not, you have an idea of the uncertainty with which the light pen can determine screen coordinates. Try refining your calibration.

Now examine the coordinate range which the light pen can measure. It should extend from (0,0) to about (319,199). After this initial adjustment, we are ready to tackle some more challenging tasks.

The first example takes advantage of the fact that the computer automatically sets some important initial parameter values whenever the instruction **USE lightpen** is invoked. This is true, for example, of the time for which the pen must be held at the same spot on the screen before its position will be registered (the procedure **delay**). This is also the case for the time which must pass from the moment when one set of coordinates has been found to the time when a new determination will begin (the procedure **timeon**). A program which is to be used to make drawings on the screen must be able to determine the coordinates of points very quickly so **delay** and **timeon** should be set to small values. If accuracy is more important than speed, then larger values should be used.

The program might look like this:

```
0010 PAGE
0020 USE lightpen
0030 USE graphics
0040 graphicscreen(0)
0050 border(0)
0060 background(14)
0070 pencolor(6)
0080
0090 xoff:=52; yoff:=-51 // use your own values
0100 offset(xoff,yoff)
0110
0120 delay(1)
0130 timeon(1)
0140
0150 REPEAT readpen(x,y,ok) UNTIL ok
0160 moveto(x,y)
0170 LOOP
0180  REPEAT readpen(x,y,ok) UNTIL ok
0190  drawto(x,y)
0200 ENDLOOP
```

Try changing the values in lines 120 and 130. What effect does this have?

Note that all lines are connected. What should be done so that the pen can be lifted and lines not connected?

If one wishes to determine the location of the pen on the text screen, the pen's coordinates must be transformed to a character position (<line>, <column>). The text screen has 25 lines each with 40 columns.

In the following example two user-defined COMAL-functions (**FUNC line(y)** and **FUNC column(x)**) are used to make the conversion. In order for the functions to operate properly, the light pen coordinates must have been corrected using the **offset** procedure described earlier, so that the lower left corner corresponds to (0,0).

The program illustrates how a light pen can be used to make selections from a menu containing characters, words or other choices. In this case the problem is to select words from the list at the end of the program and make them into a sentence with a maximum of 40 characters:

```
0010 PAGE
0020 DIM text$(25,4) OF 10
0030 DIM name$ OF 10, all$ OF 40
0040 ZONE 10
0050 l:=8
0060
0070 USE system
0080 textcolors(0,14,6)
0090
0100 arrange'words
0110
0120 USE lightpen
0130 delay(60)
0140 timeon(60)
0150 accuracy(10,2)
0160 xoff:=52; yoff:=-51 // use your own values
0170 offset(xoff,yoff)
0180
0190 choose'words
0200
0210
0220 PROC arrange'words
0230  CURSOR l,1
0240  FOR i:=1 TO 5 DO
0250   FOR j:=1 TO 3 DO
0260    READ text$(i,j)
0270    PRINT text$(i,j),
0280   ENDFOR j
0290   PRINT
0300  ENDFOR i
0310  text$(6,1):="end"
0320  PRINT text$(6,1)
0330  PRINT AT 6,1: "Point to words with the light pen."
0340 ENDPROC arrange'words
0350
0360 PROC choose'words
0370  REPEAT
0380   REPEAT readpen(x,y,ok) UNTIL ok
0390   IF y<199-(l-1)*8 THEN // from line l
```

```
0400     name$:=text$(line(y)-l+1,column(x) DIV 10+1)
0410     IF name$<>"end" THEN all$:+" "+name$
0420     PRINT AT 2,1: all$
0430    ENDIF
0440    WHILE penon DO NULL
0450   UNTIL name$="end"
0460   CURSOR 20,1
0470 ENDPROC choose'words
0480
0490 FUNC line(y)
0500   RETURN (200-y) DIV 8+1
0510 ENDFUNC line
0520
0530 FUNC column(x)
0540   RETURN x DIV 8+1
0550 ENDFUNC column
0560
0570 DATA "Peter","takes","enough"
0580 DATA "the cat","eats","from"
0590 DATA "the food","rains","always"
0600 DATA "everything","remembers","never"
0610 DATA "the book","forgets","soon"
```

In line 150 the procedure **accuracy** from the light pen package is used. The procedure **accuracy(<dx>,<dy>)** determines the resolution in the x- and y-directions.

Add some additional DATA statements yourself.

Overview of the Light Pen Package

The package contains 5 procedures and a function:

offset(<xoff>,<yoff>)
penon
readpen(<x>,<y>,<ok>)
timeon(<time>)
delay(<time>)
accuracy(<dx>,<dy>)

offset(<xoff>,<yoff>)

is a procedure which is used to offset the light pen coordinate pair so that it agrees with the corrdinates of the graphics screen. This offset can vary from display to display. Try starting with values such as: **<xoff>** = 75 and **<yoff>** =-45.

Example:

offset(52,-51)	Light pen coordinates are offset so (0,0) is in the lower left-hand corner.

penon
is a function which has the value **TRUE (=1)** if the pen is touching the screen. Otherwise **penon** equals **FALSE (=0)**.

readpen(<x>,<y>,<ok>)
is a procedure which reads the coordinates of the screen position and delivers them in the variables <**x**> and <**y**>. The variable <**ok**> has the value **TRUE** if the pen is touching the screen (just as the function **penon**).

Example:
REPEAT readpen(x,y,ok) UNTIL ok
PRINT x;y

Read the screen coordinates when the light pen is touching the screen. Print the coordinates on the next line.

delay(<time>)
is a procedure which is used to specify the **time** for which the light pen must be held still on the screen before the light pen reading will be recorded. The light pen must be held still within the limits specified in the procedure **accuracy**.

The parameter <**time**> is given in 1/60 of a second. Starting value: <**time**>=10 (i.e. 10/60 = 1/6 second)

timeon(<time>)
is a procedure which is used to specify the **time** which must pass from one screen reading until the next is possible.

<**time**> is given in 1/60 of a second. Starting value:
<**time**>=30 (i.e. 30/60 = 1/2 second)

accuracy(<dx>,<dy>)
is a procedure which is used to indicate the size of the region on the screen within which the light pen must remain to be considered to be 'at rest'. The smaller these values, the more precisely the light pen must be positioned to obtain a reading.

Initial values: <**dx**>=4 and <**dy**>=2

Example:

accuracy(10,8)	The pen is considered to be at rest if it is held within a 10x8 pixel region.

The System Package

USE system

This package contains, among other things, procedures which can be used to specify how the screen display, keyboard and printer interfaces should operate. In addition the package contains functions which provide information about your system, the display and the keyboard:

textcolors(<border>,<background>,<text>)
keywords'in'upper'case(TRUE or FALSE)
names'in'upper'case(TRUE or FALSE)
quote'mode(TRUE or FALSE)
inkey$
settime(<time'of'day$>)
gettime$
getscreen(<screen$>)
setscreen(<screen$>)
hardcopy(<unit$>)
currow and curcol
bell(<duration>)
free
defkey(<no>,<text$>)
showkeys
serial(TRUE or FALSE)
setprinter(<attributes$>)
setrecorddelay(<duration>)
setpage(<integer>)

textcolors(<border>,<background>,<text>)

is a procedure which is used to define the color combination of the border, background and text. On start-up **textcolors(14,6,14)** is executed automatically on a Commodore 64. On an SX-64 **textcolors(3,1,6)** is the default value.

Examples:

textcolors(0,2,1)	black border, red background and subsequent white text
textcolors(12,11,15)	grey tones
textcolors(-1,5,-1)	Only the background is changed (in this case to green).

keywords'in'upper'case(TRUE or FALSE)

is a procedure which determines whether keywords are to be written in upper case (TRUE) or lower case (FALSE). The default is TRUE.

Example:

keywords'in'upper'case(FALSE) Keywords are displayed in a listing with small letters.

names'in'upper'case(TRUE or FALSE)

is a procedure which determines whether names are to be written in upper case (TRUE) or not (FALSE). The default is FALSE.

Example:

names'in'upper'case(TRUE) Names will be displayed with large letters.

quote'mode(TRUE or FALSE)

is a procedure which determines whether control codes and other invisible ASCII characters in string constants are to be displayed in reverse text (TRUE) or with their ASCII values enclosed in quotes (FALSE). After start-up the default is FALSE.

Examples:

PRINT statement after **quote'mode(TRUE)**:

PRINT " ■ Hello!"

after **quote'mode(FALSE)**:

PRINT """2"Hello!"

inkey$

is a function which reads in characters from the keyboard. The function **inkey$** works like **KEY$**. However **inkey$** awaits a character with the cursor flashing at its current position.

Examples:

answer$:=inkey$

PRINT inkey$

settime(<time'of'day$>)

is a procedure which is used to set the clock in the computer (CIA#1 real time clock). On start-up the clock is zeroed by **settime("00:00:00.0")**.

The format of the **time'of'day$** string is:

hh:mm:ss.t/ff	hh is the hour (0 - 24)
or	mm is the minute (0 - 59)
hh:mm:ss	ss is the second (0 - 59)
or	t is tenths of a second (0 - 9)

hh:mm	
or	if a number field is left out,
hh	it will be assigned the value 0.
ff	is the frequency (50 or 60);
	is optional (default: 50 Hz)

SX-64 has a switchmode power supply which is 60Hz. In this case you should remember the ff parametre.

Examples:
settime("07:30:15")
settime("10:20")
settime("0") The clock is reset to 0.

gettime$

is a function which returns the time'of'day in the format hh:mm:ss.t

Examples:
PRINT gettime$ answer e.g.: 9:32:50.4

digital clock:

```
PAGE
USE system
LOOP
  PRINT AT 1,30: gettime$,
ENDLOOP
```

getscreen(<screen$>)

is a procedure which takes a copy of the current text screen, and saves it as the string **screen$**. The string **screen$** takes up 1505 characters. This is reserved by using the instruction **DIM screen$ OF 1505**.

The content of the string **screen$(1:1505)**:

screen$(1)	border color
2	background color
3	cursor color
4	cursor: line - 1
5	cursor: column - 1
6:1505	text and color information

Text and color information consists of 500 sequences of 3 bytes each:

character 1 For every two characters

character 2	their color is stored.
2. 1.	Each color takes 4 bits
color	making a byte.

See the program examples following **setscreen(<screen$>)**.

setscreen(<screen$>)

is a procedure which creates a picture on the text screen. Picture informationen is contained in the string **screen$**. The string must contain at least 1505 characters. See **getscreen(<screen$>)**.

Program example 1:

```
DIM a$ of 1505, b$ of 1505
USE system
...
...
getscreen(a$)
...
...
getscreen(b$)
a$:=a$(1:725)+b$(726:1505)
...
setscreen(a$)
```

Note:

At two selected times during the execution of the program, the contents of the text screens are saved in the strings **a$** and **b$** respectively. Later a string is created by combining the first 725 characters of **a$** (i.e. color and cursor information, and the first 12 lines of the **a$** screen image) and of **b$**'s last 780 characters (i.e. the **b$** screen's lower 13 lines). The combined image is finally presented on the screen.

Program example 2:

```
PROC help CLOSED
  DIM s1$ OF 1505,s2$ OF 1505
  USE system
  getscreen(s1$) // save screen image
  OPEN FILE 10, "screen'help",READ
  READ FILE 10:s2$
  CLOSE FILE 10
  setscreen(s2$) // shows a user help screen
  WHILE KEY$=CHR$(0) DO NULL
  setscreen(s1$) // the old image back again
ENDPROC help
```

hardcopy(<unit$>)

is a procedure which prints out the contents of the text screen to the unit which is given in the string **unit$**. The printout begins with a carriage return.

Example:

hardcopy("lp:")	The contents of the text screen is printed on a lineprinter. The instruction has the same effect as <**CTRL-P**>.

currow and ***curcol***

are two functions which return the current row and the current column respectively.

Examples:
row:=currow; column:=curcol
PRINT AT 0,curcol-5: name$

bell(<duration>)

is a procedure which activates COMAL's "bell". The parameter **duration** must be an integer in the range 1 to 255. The value **1** corresponds to a real-time duration of about 0.15 seconds. On start-up an automatic **bell(3)** is executed.

Example:

bell(10)	Sound for 1.5 sec.

free

is a function which returns the number of free bytes in working memory. A more complete overview of the use of working memory is obtainable using the command SIZE. But because SIZE is a command, it cannot be used from a running program.

Example:
PRINT free

defkey(<no>,<text$>)

is a procedure which is used to redefine the meaning of the function keys. The keys are numbered 1,..,8,11,..18.

The numbers 1 - 8 are normally active for indication of the usual function keys <f1> - <f8>. But during program execution, the function keys will correspond to numbers 11 - 18. The string **text$** may consist of a maximum of 32 characters.

The procedure **showkeys** will print out a list of the current definitions of the function keys.

Examples:
On start-up the following is performed:

defkey(6,"LIST ") Activating <f6> prints LIST on the screen.

The <f3> and <f4> can e.g. after redefinition be used to assist with the writing of procedures:

defkey(3,"AUTO"13""13"PROC ")
defkey(4,"ENDPROC"13""141"SCAN"13"")

<f3> will cause: AUTO
0010
0020 PROC

<f4> will cause: XXXX ENDPROC
(Interrupt AUTO-numbering.)
SCAN
(Which checks the structure of the procedure and allows use of the procedure as a command.)

Program example:

```
USE system
defkey(15,"COMAL for everyone!"13"")
INPUT "What did you say? ": text$
PRINT text$
```

If the <f5> key is activated in response to the INPUT statement, the system will react as if the message came from the keyboard and print it out.

showkeys

is a procedure which controls whether communication is sent to the serial port or to the Commodore IEEE-488 module (if available).

serial(TRUE or FALSE)

is a procedure which controls whether communication is sent to the seria port or to the Commoore IEEE-488 module (if available).

Examples:

serial(TRUE) Send to the serial port.
serial(FALSE) Send to the IEEE-488 module.

setprinter(<attributes$>)

is a procedure which is used to select the unit number and attributes of the peripheral printer. Printout to the lineprinter (**lp:**) will thereafter be performed according to the rules given by the attributes. These are given in a string during procedure calls.

Possible printer attributes:

/a-	do not translate from C64 ASCII to standard ASCII
/a+	convert from C64 ASCII to standard ASCII
/l-	suppress line feed after carriage return
/l+	execute line feed after each carriage return
/t-	ignore 'time out' signal and continue printout
/t+	interrupt with error message if time runs out

Secondary adresses for the Commodore MPS 801 (partly also MPS 802) printer: (See instruction manuals for other printers.)

/s-	no secondary address used
/s0	write data as received
/s1	write data in previously defined format
/s2	save format information
/s3	number of lines per page
/s4	allow explanatory error messages
/s5	define a programmable character
/s6	number of blank lines between each printed line
/s7	print with lower case

Upon start-up in COMAL "**lp:**" is defined as the unit with the attributes **u4:/a-/l-/t+/s7**.

The MPS 801 printer can be set to act as unit 4 or unit 5 by means of a switch on the back panel.

Examples:

setprinter("u5:/s0")	"lp:" means hereafter unit 5; printout with upper case.
setprinter("lp:/a+/l-")	Convert to ASCII, send no line feed.

A procedure to define the number of lines per page on the MPS 802 printer:

```
PROC page'802(lines'pr'page) CLOSED
  OPEN FILE 1,"lp:/s3",WRITE
  OPEN FILE 2,"lp:",WRITE
  PRINT FILE 1: CHR$(lines'pr'page),
  PRINT FILE 2: CHR$(147),
  CLOSE
ENDPROC page'802
```

setrecorddelay(<duration>)

is a procedure which causes COMAL to pause during writes to a random file. The parameter <duration> is given in milliseconds. The disk operating system needs time to write a block to the diskette before the COMAL system can send a new positioning instruction. It is rarely necessary to use the procedure. When COMAL is initiated, an automatic **setrecorddelay(50)** is carried out, unless the Commodore IEEE module is connected with the COMAL cartridge. In that case a **setrecorddelay(0)** will be executed.

setpage(<integer>)

is a procedure which determines to which overlay the instructions PEEK and POKE will refer. See Chapter 8 for more information on this. The utilities program **showlibs** on the demo diskette (or tape) uses this procedure.

Examples:

setpage(0)	**$8000 - $9fff**	**RAM used by packages**
	$A000 - $bfff	**hidden RAM used by packages**
	$d000 - $dfff	**hidden RAM**
	$e000 - $ffff	**graphic screen**

The Font Package

The package **font** contains 6 procedures which are used to define new screen characters. It is possible to change an entire character set or just an individual character.

The package is activated with the order

USE font

The package affects the 4 character sets numbered:

0: User-defined	read/write
1: User-defined	read/write
2: Upper case/graphics set	read only
3: Upper/lower case letters	read only

The Commodore 64 uses a double character set. Normally COMAL uses character set 3. By activating <**SHIFT C=**> you can switch back and forth between character sets 2 and 3. These two character sets are permanently available in the memory of the computer, so they cannot be changed.

With the font-package it is possible to add a new double character set numbered 0 and 1. This character set is stored in a protected area of the working memory of the computer.

There are now several options:

1. You can move a copy of the normal character set of the computer into the area reserved for the user character set and change some of the characters.
2. You can fetch a completly new character set from diskette or tape and store it as the user-defined character set. It will go into effect at once. Of course it is essential to have such a character set prepared and available on diskette or tape. A character set is available on your demo diskette or tape. But it is also possible to create your own and to store it for later use.

Remarks:

* The character set used corresponds to -screen- characters. Their character code are not in accord with the standard ASCII values. See Appendix A for standard screen character codes and ASCII codes.
  The following command will print out all the standard screen characters. Issue the order with default screen and cursor colors. (The screen image starts at memory address 1024.):
  for I=0 to 255 poke 1024+i,i
* The user-defined character set is also used by the procedure **plottext** from the graphics-package.
* Because a printer uses its own character set, **font** will have no effect

on PRINT and LIST orders directed to the printer. On the other hand <CTRL-D> (**printscreen**) will cause an exact copy of the graphics screen image to be printed out on a MPS 801 compatible printer.

Example of character replacement:
First we fetch a character from the standard character set to see how it is stored in an 8x8 raster pattern of pixels. The following program can be used for this purpose.

The character is fetched by means of the procedure **getcharacter**. The rest of the program has been added to provide a nice printout of the character in an 8x8 matrix. We let the string function **bin$** convert the individual characters in the fetched raster pattern to binary numbers. These numbers are then printed under one another to create the bit pattern of the character:

```
0010 // save "Fetch Character"
0020 USE font
0030 DIM raster$ OF 8
0040 PAGE
0050 INPUT "Character set  : ": choice#
0060 INPUT "Character no.  : ": character#
0070 PRINT
0080 getcharacter(choice#,character#,raster$)
0090 FOR i:=1 TO 8 DO
0100   PRINT TAB(12),bin$(ORD(raster$(i)))
0110 ENDFOR i
0120
0130 FUNC bin$(number) CLOSED
0140   DIM binnumber$ OF 8
0150   binnumber$:="00000000"
0160   bit:=1
0170   FOR i#:=8 TO 1 STEP -1 DO
0180     IF number BITAND bit THEN binnumber$(i#):="1"
0190     bit:+bit
0200   ENDFOR i#
0210   RETURN binnumber$
0220 ENDFUNC bin$
```

Try out the program. Choose a character from the double standard character set: 2 or 3. Since we have not yet prepared any user-defined characters, any attempt to fetch a character from sets 0 or 1 will result in an error message. The character **a** has the number **1** in character set number **3**.

Next we will prepare a user-defined character by simply moving a copy of the standard character set up to the user-defined area. Since no user-defined character set has yet been created, the order **linkfont** has this effect. Write therefore:

```
use font
linkfont
```

This way a user-defined character set (0 and 1) is created for immediate use. In addition the old screen image is hidden, and a new picture is created for using the new character set. It is always possible to return to the standard character set by using the order DISCARD (which preserves any program in working memory) or NEW (which does not).

It is now possible to change the characters in the double character set 0 - 1. The brief program which follows reads in individual characters by means of DATA statements and makes them part of the new character set with the order **putcharacter**.

The DATA in the program is for a letter ø (the Greek letter 'phi'). This character can replace any character in set 0 or 1. If you wish to have this Greek letter available instead of the "pound" sign, you can assign it character number **28** in character set number **1**. Then when you press the "pound" key, a ø will appear on the screen.

Try replacing some other characters. Notice that there is an immediate effect on the display screen.

```
0010 // save "Save Character"
0020 USE font
0030 DIM raster$ OF 8
0040 FOR I:=1 TO 8 DO
0050   READ byte
0060   raster$(i:i):=CHR$(byte)
0070 ENDFOR i
0080 PAGE
0090 INPUT "Character set  : ": choice#
0100 INPUT "Character no.  : ": no#
0110 putcharacter(choice#,character#,raster$)
0120
0130 DATA %00000000
0140 DATA %00000000
0150 DATA %00111110
0160 DATA %01101110
0170 DATA %01111110
0180 DATA %01110110
0190 DATA %01111100
0200 DATA %00000000
```

(NB: Remember to execute **use font** and **linkfont** before running this program.)

Replacing an entire character set

If a double character set is available on diskette, it can be fetched into working memory by using the orders:

discard (to erase earlier **linkfont**)
use font
loadfont(<filename$>)

where **<filename$>** is the name of the diskette- or tape file. Thereafter the new character set and screen image can be used.

The package font contains the procedures:

linkfont
loadfont(<filename$>)
savefont(<filename$>)
keepfont
getcharacter(<character set>,<character>,<raster$>)
putcharacter(<character set>,<character>,<raster$>)

Font Package Procedures in Depth

linkfont

is a procedure which is used to define a new double character set number 0 and 1. The procedure should only be used as a direct command, for a program cannot continue after a **linkfont** statement.

* Room is reserved in working memory for the new character set and for the screen image (4000 bytes for the character set and 1000 bytes for the screen image).
* The extra screen becomes the current screen and is cleared.
* Because the variable table in the working memory is overwritten by the new character set, all COMAL and package names will be undeclared.
* If **linkfont** has not been called earlier, either directly or indirectly through **loadfont**, then the standard character set (2-3) will be copied over as the new character set (0-1).
* If **linkfont** has been called previously, nothing happens. It is thus not possible to overwrite an existing user-defined character set with a new **linkfont**-command. The user-defined character set must be removed first by using the order DISCARD or NEW. Individual characters on the other hand can be replaced using the order **putcharacter**.
* The double character set is treated as a part of your COMAL program. When the program is stored using the SAVE command, the user character set is saved along with it as a single file. When the program is loaded again later using the LOAD order, the character set is also loaded and ready to go (even before the program is run!).

loadfont(<filename$>)

is a procedure which reads in a character set with the name **<filename$>** from diskette or cassette tape. First **loadfont** executes an automatic **link-**

font, reserving room in working memory for the fetched character set and the extra screen image.

The procedure **loadfont** replaces any existing user character set with the one which has been read in. The new character set and screen image can then be used as the current character set and screen.

savefont(<filename$>)

is a procedure which copies the user-defined double character set from the working memory and saves it on diskette or tape under the name **<filename$>**.

keepfont

is a procedure which is used to "freeze" a user-defined character set, so that it cannot be deleted using DISCARD or NEW. It is necessary to turn off the computer to return to the standard character set.

- **loadfont** still works. A newly read-in character set will also be "frozen".
- After **keepfont**, characters will NOT be saved together with a COMAL program by the command SAVE.

getcharacter(<character set>,<character>,<raster$>)

is a procedure which fetches a raster image of the character with the screen code **<character>** from **<character'set>**. The image is fetched in the form of a string variable **<raster$>** which is 8 characters long.

Permitted values:
<character sets>: 0, 1, 2 or 3
<characters> : 0, 1,..., 255

Examples:

getcharacter(3,1,raster$) The character **a** is fetched from character set 3.

```
DIM a$ OF 8
USE font
getcharacter(2,4,a$)
PRINT a$
```
The code character **D**'s raster image is fetched and displayed.

Printout: **XLFFFLX**

putcharacter(<character set>,<character>,<raster$>)

is a procedure which assigns the character with the screen code **<cha-**

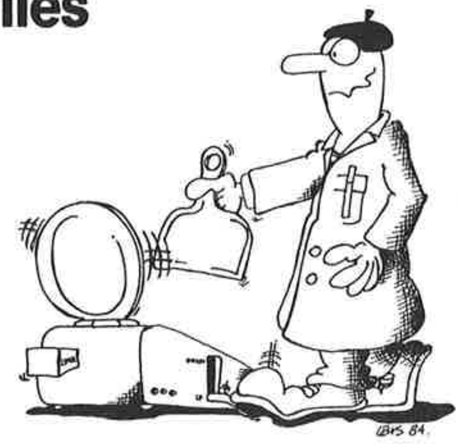
racter> in **<character set>** with the raster pattern in the string **<raster$>**.

Allowed values:
<character'set>: 0, 1
<character> : 0, 1,..., 255

Examples:
putcharacter(1,5,""0""0"<FFF<"0"")
In the extra character set **1** the character **o** is assigned screen character number **5**.

Chapter 6

COMAL Files



What is a file?

As you begin to use your computer to do more and more jobs for you, it will be very convenient to be able to create files for storing information. You may wish to save business transactions, financial records, address lists, the results of calculations, measurements or other data for later use. Of course it is possible to purchase commercially produced "database" software to help you do this. Nevertheless many computer owners elect to write their own programs, so that they can tailor them precisely to their particular needs.

A *file* is a collection of data, organized for storage and retrieval. The *storage medium* can be a Datassette *tape* or a *diskette*. Because serious file handling usually requires the use of a disk drive, this chapter will concentrate mostly on file storage with a disk drive.

There are several ways in which files can be organized. Sometimes it is convenient to save a set of information one item after the other in a *sequential file*. Sequential files are easy to use and do not require a great deal of prior planning with respect to the number of storage units each item will require. On the other hand when sequential files are used it is necessary to read the entire file into the computer's memory. And you must re-save the entire file again each time you have finished working with it. Storing data as sequential files is useful as long as the file does not get too large.

There is a way to get around the problem of having to handle the entire file all at once. *Random-access files* can be created, so you only need to read in a small portion of the file when you want to change it or refer to it.

In this case you must plan ahead carefully, allotting an appropriate amount of space for each "set" of data (i.e. each *record*). If we know how much room each record takes up, it is possible to fetch or save a single record at a time. Thus the use of random-access files can speed up access to some types of information on a diskette (they cannot be used with a Datassette tape unit). Random-access files are appropriate to use for handling large collections of systematic data.

In this chapter we intend to cover several important uses for COMAL files:

* saving and loading *programs* and *procedures*
* an *address list* filing program using sequential files
* a random-access *inventory file* program
* *moving files* between diskettes

The demonstration programs described in this chapter are found on the demo diskette (or tape) distributed with the COMAL cartridge. In addition, complete program listings are available in Appendix H.

Saving Programs and Procedures

As you proceed to write more programs, you will find that they become larger. But you will also find that many of the operations to be carried out are the same: saving data, fetching data, printing tables, printing a title screen, entering a user response from the keyboard, etc. It will become very natural for you to do these jobs and others which may be required again by using COMAL procedures. Later on the same procedures can be used with little or no changes.

There are a number of COMAL disk drive operations which make the building of new programs from available procedures particularly easy and convenient to do. An overview of these operations is shown in the table which follows.

	COMPLETE PROGRAMS	PROGRAM SEGMENTS
STORAGE:	SAVE "<file name>" LIST "<file name>"	LIST<segment>"<file name>"
	SAVE "testfile" LIST "testfile"	LIST 1000-1095 "printout" LIST printout "printout.1"
RETRIEVAL:	LOAD "<file name>" ENTER "<file name>"	MERGE<line>"<file name>" ENTER "<file name>"
	LOAD "testfile' ENTER "testfile"	MERGE "printout" MERGE 1100 "printout" MERGE 1100,5 "printout" ENTER "printout"

It should also be mentioned here that the DISPLAY command can be used to save entire programs, individual procedures or sequences of line numbers to diskette (or tape). The instruction **DISPLAY 10-100 "sample"** saves program lines 10-100 **with no line numbers** as an ordinary sequential file. The file can (only) be retrieved using an INPUT FILE instruction or the GET$ instruction. Other formats, analogous to the formats of the LIST command, are also permitted. The DISPLAY command can be used to create a sequential file from a COMAL program. The file might then be loaded into a text editor (e.g. EASYSCRIPT).

Let's quickly review the storage and retrieval of COMPLETE PROGRAMS on disk. Consider the lefthand column of the table.

The commands SAVE and LOAD are already familiar to you. You can use SAVE to transfer a copy of your COMAL program file currently in memory to diskette. The syntax is **SAVE "<file name>"**, where the item **<file name>** is a program name (up to 16 characters) of your choice. Beginning a file name with @ has a special meaning: The file will be deleted if it exists, and the new file will be saved in its place under the same file name.

The instruction LOAD is the reverse operation of SAVE. The LOAD instruction has the format **LOAD "<file name>"** and causes the program file **<file name>** to be copied from the diskette to the COMAL program storage memory of your Commodore 64. When you LOAD a program file, the previous contents of the COMAL program area will be erased. Note that only a program file (denoted by **prg** in the directory) can be LOADed. A complete program can also be LISTed to a diskette where it will be stored as a sequential file. It must then be ENTERed or MERGEd to be retrieved later.

The commands RUN and CHAIN can also be used to bring program files into memory from the disk drive. See the more detailed description of these instructions in Chapter 4. In addition a *closed procedure*, saved using SAVE, can be fetched as an *external procedure* during program execution (see the descriptions of EXTERNAL and PROC in Chapter 4).

Now let's take a look at the column titled PROGRAM SEGMENTS. This information can be a real time-saver, so study it carefully!

Suppose that you have developed an ingenious procedure called **quick'save** for quickly storing a list of items and prices on diskette. The procedure is so general, that it could be useful in many other programs. If it is more than a few lines in length, it will not be convenient to type it in each time it is to be used. It should be LISTed to diskette by using the instruction **LIST quick'save "prc.quick'save"**. If you do this and type **dir**, you will observe that the file **prc.quick'save** is stored as a sequential (not a program) file on the diskette. Note that it cannot be LOADed like a program file; to get it into program memory you must use either MERGE or ENTER. These instructions will be described shortly.

You are also permitted to save your procedure by referencing the line numbers for the procedure. For example you could store **quick'save** by writing: **list 1000-1090 "prc.quick'save"**, if the line numbers are correct. Typing **dir** will verify that the procedure has been saved as a sequential file. The procedure can now be brought into another program using the MERGE or ENTER instructions.

Now comes the good part. When you want to use the procedure again, you have the following alternatives:

* Write **merge "prc.quick'save"**. Your procedure will be copied from the disk drive and appended to the program in memory. It will appear with line numbers starting 10 beyond the last line number in the program, even though the original file was LISTed with line numbering 1000-1090
* You may instead choose to write **merge 1100 "prc.quick'save"**. In this case the procedure will appear in your current program from lines 1100 and beyond with a line number interval of 10 (the default value). If you want the procedure to start at line 1100 with an interval of 5 between lines, just write **merge 1100,5 "prc.quick'save"**.

WARNING: Be careful when merging a procedure in the middle of a program. You must be sure that there is room enough for the procedure with the line number interval selected. Otherwise the procedure will get mixed up with other instructions or erase them if line numbers coincide.

* In case you have LISTed an entire program to diskette, you may want to use the instruction ENTER. If you write **enter "printout"** for example, the sequential file **printout** will be read into the active program area. (NOTE: Any other program in memory will be deleted.)

If you have worked with other programming languages and operating systems, you will appreciate how convenient these facilities can be while developing programs.

Sequential Files - An Address List

In Chapter 3, **program 19**, we saw a simple example illustrating how to save numbers on a sequential file. You may recall that a sequential file must be opened before data is saved or fetched. After use the file must be closed.

The formal structures for these operations are as follows:

Open a file, save data, close the file.

```
OPEN FILE <fileno>,<filename$>,WRITE
...
...
PRINT FILE <fileno>: <data element>
...
...
CLOSE FILE <fileno>
```

Open a file, fetch data, close the file.

```
OPEN FILE <fileno>,<filename$>,READ
...
...
INPUT FILE <fileno>: <variable name>
...
...
CLOSE FILE <fileno>
```

We will now turn our attention to a practical problem. Suppose you want to create a program to save names, addresses and telephone numbers. The following example illustrates the kind of data we want to save and variable names which we will use:

example	***string variable***
John Smith	**name$()**
1200 Wilson Drive	**street$()**
Anytown, PA 19380	**town$()**
(212) 123-4567	**phone$()**

Notice that all four string variables are to be defined as arrays. We intend to design the program to handle up to 100 names with addresses and phone numbers. We will refer to the collection of information illustrated above a **data record** and each of the individual variables which constitute the record as a *data element*. In this example each data record will consist of four elements.

Note that all four string variables must be declared as one dimensional arrays. We plan to permit our program to handle up to 100 records. Consider some of the tasks which this program will have to handle:

* LOAD all the records in the data file into memory
* CREATE a data record with name, address and phone number
* LIST all records in the file
* SEARCH through the file to find certain records
* SORT the file alphabetically by name
* CHANGE a record
* DELETE a record
* SAVE the file on diskette

Of course there are other operations one might want to perform on a file, but we will limit this example to the above operations in the interest of simplicity. When you have understood the procedures described in this section, you will be able to extend or revise this program so that it best suits your needs. Those of you who received this book with the COMAL cartridge will find this program on the demo diskette or tape under the file name **Addr List Demo**. The program listing is also given in Appendix H.

We intend to take a careful look at this program. Please note that it has not been "optimized". It has been written as simply and clearly as possible to make it easy to understand. As you learn more and more about sequential files, feel free to make modifications and improvements!

The program starts out with a line indicating the name of the file:

```
0010 // SAVE "@0:Addr List Demo
```

Notice that we have used a remark statement (//) and included **SAVE "** ahead of the file name. This little trick makes it easier for you to save modifications of your program as you develop it and revise it. Just move the cursor to this line, remove the first part of the line by typing blanks (or use <INST/DEL>). When you press <RETURN>, the new version will be saved. The @ symbol included as the first character of the file name causes the existing program file to be deleted before the new file is saved.

The drive designation **0:** should always be included to avoid problems after many save operations to the same diskette.

WARNING: Be careful when using this method; you could lose a program file. Be sure you have a backup copy of your program and update it from time to time. Do not make revisions using the demonstration diskette or tape. Load the program, then save later revisions to another storage disk or tape.

The next lines in the program listing take care of DIMensioning of arrays and string variables used in the program:

```
0020 DIM reply$ OF 1, name$(100) OF 40
0030 DIM street$(100) OF 40, city$(100) OF 40
0040 DIM phone$(100) OF 20, flag$ OF 40
0050 DIM searchkey$ OF 40, string$ OF 150
0060 number:=0 // number of records
```

Notice here that we have made provision for the storage of 100 data records each consisting of four elements: a name, street, town and phone number. Each of these elements may be up to 40 characters long. This choice means that the sequential file can take up to 4x40x100 or about 16 kilobytes in memory. Since a total of about 30 KB is available, and the program only takes up about 4 KB, more room is available. You can change these numbers, if you wish.

Next comes an **introductory screen** describing the program:

```
0070 PAGE
0080 PRINT "This program illustrates the use of"
0090 PRINT "SEQUENTIAL FILES. It can be used to"
0100 PRINT "create a list of names, addresses"
0110 PRINT "and telephone numbers."
0120 PRINT "Each record will have the format:"
0130 PRINT
0140 PRINT "      name"
0150 PRINT "      street"
0160 PRINT "      city"
0170 PRINT "      phonenumber"
0180 PRINT
0190 PRINT
0200 PRINT "Press any key to continue..."
0210
0220 wait'for'keystroke
0230
```

The statement PAGE clears the screen and the following lines simply print information on the screeen. Notice the procedure **wait'for'keystroke**. This is a procedure which you might find convenient to use in your own programs:

```
2240 PROC wait'for'keystroke
2250   PRINT
2260   PRINT "< >...";
2270   REPEAT
2280     reply$:=KEY$
2290   UNTIL reply$<>CHR$(0)
2300   PRINT AT 0,2: reply$
2310 ENDPROC wait'for'keystroke
```

You may or may not want < >... to be printed on the screen whenever the computer is awaiting an operator response. Change it or delete it as you wish. The REPEAT...UNTIL loop will be executed continuously as long as no key is depressed, since the value of the COMAL function KEY$ remain equal to CHR$(0). When a key is pressed, KEY$ takes on the value of the character sent from the keyboard, and **reply$** will no longer be equal to CHR$(0). The REPEAT...UNTIL loop will be terminated, and execution proceeds to the next line. The **PRINT AT 0,2: reply$** statement causes the character which was sent from the keyboard to appear inside the brackets in the < >... symbol.

Now take a look at the **main program loop**:

```
0240 LOOP
0250   show'menu
```

```
0260   flag$:=""
0270   wait'for'keystroke
0280   CASE reply$ OF
0290   WHEN "1"
0300     load'file
0310   WHEN "2"
0320     create'record
0330   WHEN "3"
0340     list'file
0350   WHEN "4"
0360     search'file
0370   WHEN "5"
0380     sort'file
0390   WHEN "6"
0400     change'record
0410   WHEN "7"
0420     delete'record
0430   WHEN "8"
0440     save'file
0450   OTHERWISE
0460     PRINT "Illegal reply.."
0470     wait'for'keystroke
0480   ENDCASE
0490 ENDLOOP
```

This is really the heart of the program. The first subprocedure encountered displays the program menu:

```
0500
0510 PROC show'menu
0520   PAGE
0530   PRINT "-----===== MAIN MENU =====-----"
0540   PRINT
0550   PRINT
0560   PRINT "    <1> LOAD   the file"
0570   PRINT "    <2> CREATE a record"
0580   PRINT "    <3> LIST   the file"
0590   PRINT "    <4> SEARCH the file"
0600   PRINT "    <5> SORT   alphabetically"
0610   PRINT "    <6> CHANGE a record"
0620   PRINT "    <7> DELETE a record"
0630   PRINT "    <8> SAVE   revised file"
0640   PRINT
0650   PRINT
0660   PRINT "Records: ";number
0670   IF number=0 THEN flag$:="Please load or create file..."
0680   PRINT
0690   PRINT flag$
0700 ENDPROC show'menu
```

The procedure clears the screen, indicates the user choices available, and shows the number of records in the file. The string variable **flag$**, which is used in the program to inform the user about various conditions, will be set equal to **Please load or create file...** if there are no records in memory. This message will be printed below the menu to guide the user.

Considering again the **main program loop**, we see that the variable **flag$** is again set equal (in line 260) to the empty string, so it can be used later for other purposes. In the next line the procedure **wait'for'keystroke** is executed. When a valid user choice has been entered (a digit from 1 to 8), the program will branch as appropriate . If the choice is not a valid one, the program simply prints out the message **Illegal reply..** and waits for you to press any key.

We will now consider each of the eight available file handling functions. The first user choice, activated by selecting **1** from the menu, is LOAD the file:

```
0710
0720 PROC load'file
0730   OPEN FILE 1,"Addresses",READ
0740   INPUT FILE 1: number
0750   FOR no:=1 TO number DO
0760     INPUT FILE 1: name$(no)
0770     INPUT FILE 1: street$(no)
0780     INPUT FILE 1: city$(no)
0790     INPUT FILE 1: phone$(no)
0800   ENDFOR no
0810   CLOSE FILE 1
0820 ENDPROC load'file
0830
```

Of course this procedure can only be used after a file has been created and is available on the diskette in the disk drive. Usually this will be first choice a user makes after starting the program. It is important to understand the procedure **load'file**, for it shows you how to read a sequential file from a diskette into program memory. The first thing done in this procedure is to OPEN FILE number **1** as a READ file called **Addresses**. Should you want to call the file some other name, you can simply alter this file name to one of your choice, here and elsewhere in the program. The easiest way to do this is by means of the CHANGE instruction (**change "Addresses","your choice"**).

We have decided to let the first element in the file be called **number** corresponding to the number of records in the file. This variable was the first one to be saved, and it is the first one to be read in now. Now that the number of records in the file is known, the file itself can be read in using a simple FOR...ENDFOR loop. (In Chapter 3 we used the logical function EOF(<fileno>) to announce End Of File).

Notice the use of the INPUT FILE statement to define the elements in the arrays **name$()**, **street$()**, **town$()** and **phone$()**. Finally, notice that **file 1** must be CLOSEd after the data input is completed.

The following procedure, activated by user choice **2**, can be used to create new records for the file:

```
0840 PROC create'record
0850   PAGE
0860   PRINT ":::: CREATE A NEW RECORD ::::"
0870   PRINT
0880   PRINT
0890   IF number=100 THEN flag$:="No more room for data!"
0900   IF flag$="" THEN
0910     number:+1
0920     INPUT "Name   ": name$(number)
0930     INPUT "Street ": street$(number)
0940     INPUT "City   ": city$(number)
0950     INPUT "Phone  ": phone$(number)
0960   ENDIF
0970 ENDPROC create'record
0980
```

The procedure begins by clearing the screen and indicating to the user what is happening. If there is no more room for data (because **number = 100**), then the message variable **flag$** is set to **No more room for data!**, and the next lines will not be executed (the condition **flag$=""** will not be fulfilled). If there are fewer than 100 records, then the number of records counter **number** will be updated (**number:+1** in line 910), and the user can input the four data elements. Execution returns to the main program loop.

User choice **3** allows entire contents of the file to be listed:

```
0990 PROC list'file
1000   PAGE
1010   PRINT ":::: LISTING THE FILE ::::"
1020   PRINT
1030   IF number=0 THEN
1040     flag$:="No files in memory!"
1050     PRINT
1060   ELSE
1070     FOR no:=1 TO number DO print'record(no)
1080   ENDIF
1090 ENDPROC list'file
1100
```

The screen is cleared and a user message is displayed. If there is no file in memory (**number—0**), then a message is sent back to the menu display by means of **flag$**. If there is a file in memory, then it is listed by the FOR...ENDFOR loop. Notice that a wait occurs as each record is displayed. Simply holding down any key will make the records scroll up the screen.

The **search** option is activated by user choice **4** from the main menu. When a file is available in memory, it can be searched find a name, street, town or other information:

```
1110 PROC search'file
1120   PAGE
1130   PRINT ":::::  FILE SEARCH :::::"
1140   PRINT
1150   PRINT
1160   flag$:="I am searching..."
1170   INPUT "Search key: ": searchkey$
1180   FOR no:=1 TO number DO
1190     string$:=name$+street$(no)+city$(no)+phone$(no)
1200     IF searchkey$ IN string$ THEN print'record(no)
1210   ENDFOR no
1220   flag$:=""
1230 ENDPROC search'file
1240
```

After clearing the screen and informing the user, this procedure allows a *search key* to be entered. This can be any string of characters at all, however capitalization must be the same as in the record element which is to be searched for. The COMAL statement **IF <condition> THEN <procedure>** is most useful here. Each record is checked by means of the FOR...ENDFOR loop. If any record contains the search key, then the entire record will be printed by the subprocedure **print'record(nr)**:

```
1250 PROC print'record(no)
1260   PRINT
1270   PRINT AT 0,10: "---------------(",no,")"
1280   PRINT AT 0,10: name$(no)
1290   PRINT AT 0,10: street$(no)
1300   PRINT AT 0,10: city$(no)
1310   PRINT AT 0,10: phone$(no)
1320   PRINT
1330   IF flag$="I am searching..." THEN wait'for'keystroke
1340 ENDPROC print'record
1350
```

Now we will examine one of the more challenging procedures in this program. The entire file can be sorted alphabetically by name. This option is activated by user choice **5** from the menu. Note that a prerequisite for proper use of this function is of course that names must be entered correctly, last name first, as the first element of each record. Of course a sort could be carried out according to any other element you may choose by simply modifying the procedure which follows as appropriate.

```
1360 PROC sort'file
1370   PAGE
1380   PRINT ":::::  SORT BY NAME ALPHABETICALLY :::::"
1390   PRINT
1400   PRINT
1410
1420   PROC swap(REF a$,REF b$) CLOSED
1430     c$:=a$; a$:=b$; b$:=c$
```

```
1440   ENDPROC swap
1450
1460   REPEAT
1470     no'swap:=TRUE
1480     FOR no:=1 TO number-1 DO
1490       PRINT AT 10,1: "Sorting... ",no
1500       IF name$(no+1)<name$(no) THEN
1510         swap(name$(no),name$(no+1))
1520         swap(street$(no),street$(no+1))
1530         swap(city$(no),city$(no+1))
1540         swap(phone$(no),phone$(no+1))
1550         no'swap:=FALSE
1560       ENDIF
1570     ENDFOR no
1580   UNTIL no'swap
1590 ENDPROC sort'file
1600
```

As in Chapter 3 **Program 19**, the sorting algorithm used here is the simple **bubble sort**. Compared to what we did in Chapter 3, we have placed the **swap** procedure inside the **sort'file** procedure. This is done to show an example of a local procedure inside another procedure.

The FOR...ENDFOR loop is carried out for each pair of names in the list. If the names are not in alphabetical order the names are swapped. The variable **no'swap** will now be equal to **FALSE**, if a swap has occurred. The **REPEAT...UNTIL no'swap=TRUE** loop is repeated until no two names are swapped on a pass through the list. The bubble sort is not the most efficient sorting technique, but it is perhaps the easiest to understand. On the demo diskette you will find a **quick'sort** procedure which is much more efficient but harder to understand.

It will sometimes be necessary to change the contents of a record in the file. This choice is activated by selecting **6** from the menu. The procedure **change'record** is shown below:

```
1610 PROC change'record
1620   PAGE
1630   PRINT "::::: CHANGE A RECORD :::::"
1640   PRINT
1650   PRINT
1660   INPUT "Which record number? ": no
1670   IF no<=number THEN
1680     print'record(no)
1690     INPUT AT 14,1: "Right record ? (y/n)? ": reply$
1700     PRINT
1710     PRINT
1720     IF reply$ IN "yY" THEN
1730       INPUT "Name   : ": name$(no)
1740       INPUT "Street : ": street$(no)
1750       INPUT "City   : ": city$(no)
1760       INPUT "Phone  : ": phone$(no)
1770     ENDIF
```

```
1780   ELSE
1790     flag$:="There are only "+STR$(number)+" records"
1800   ENDIF
1810 ENDPROC change'record
1820
```

The procedure should be easy to follow. It involves simply requesting the user to indicate which item is to be changed then allowing the change to be entered. Notice again the use of the variable **flag$** to transmit an error message to the menu.

Selecting user option **7** from the menu allows a record to be **deleted**. A procedure which can accomplish this function is as follows:

```
1830 PROC delete'record
1840   PAGE
1850   PRINT ":::: DELETE A RECORD ::::"
1860   PRINT
1870   PRINT
1880   INPUT "Which record number? ": record
1890   IF record>number THEN
1900     flag$:="Use a smaller record number!"
1910   ELSE
1920     print'record(record)
1930     PRINT
1940     INPUT "Is this the right record (y/n)? ": reply$
1950     PRINT
1960     IF reply$ IN "yY" THEN
1970       FOR no:=record TO number-1 DO
1980         name$(no):=name$(no+1)
1990         street$(no):=street$(no+1)
2000         city$(no):=city$(no+1)
2010         phone$(no):=phone$(no+1)
2020       ENDFOR no
2030       number:-1
2040     ENDIF
2050   ENDIF
2060 ENDPROC delete'record
2070
```

After a file has been entered, sorted or modified, it will usually be desirable to save it for later use. Choose user option **8** from the menu to activate the following procedure:

```
2080 PROC save'file
2090   PAGE
2100   PRINT ":::: SAVING FILE TO DISK ::::"
2110   OPEN FILE 1,"@0:Addresses",WRITE
2120   PRINT FILE 1: STR$(number)
2130   PRINT
2140   PRINT
2150   FOR no:=1 TO number DO
2160     PRINT FILE 1: name$(no)
```

```
2170      PRINT FILE 1: street$(no)
2180      PRINT FILE 1: city$(no)
2190      PRINT FILE 1: phone$(no)
2200    ENDFOR no
2210    CLOSE FILE 1
2220 ENDPROC save'file
2230
```

To save the file, the file must first be opened, indicating the number of the file, **1** in this case, the file name, in this case simply **Addresses**, and the fact that the file is opened as a WRITE file. Of course you can alter this procedure to make it possible to make the file name user selectable. Just insert an input statement like **INPUT "File name? ":filename$** early in this procedure. You will also have to change the procedure **load'file** to allow user choices there too.

The procedure **save'file** continues by first saving the number of records in the file (**PRINT FILE 1: STR$(number)**). This information is the first thing to be read in when the file is loaded again. The PRINT FILE statements are used to transmit the contents of each record to the sequential file. Finally the file must be CLOSEd.

Random Access Files - an Inventory Program

To illustrate the use of *random access files* (also called *direct files*), we will describe a simple inventory program. The program **Random File Demo** can be found on the demo diskette. You may wish to try LOADing, RUNning and LISTing the program before continuing.

The first few lines of the program identify it (and facilitate saving) and DIMension the string variables to be used:

```
0010 // save "@0:Random File Demo"
0020 DIM code$ OF 30, part$ OF 30
0030 DIM quantity$ OF 30, price$ OF 30
0040 maxquantity:=25
```

Next comes a brief description of the program, displayed as soon as the program is RUN:

```
0050 PAGE
0060 PRINT "::: RANDOM FILE DEMONSTRATION ;;;"
0070 PRINT
0080 PRINT
0090 PRINT "This program illustrates as simply as"
0100 PRINT "possible how you can store and retrieve"
0110 PRINT "information from a 'direct' or"
0120 PRINT "'random-access' file."
0130 PRINT
0140 PRINT "This example serves to save and retrieve"
0150 PRINT "information about a parts inventory"
```

```
0160 PRINT
0170 PRINT "The information is arranged:"
0180 PRINT
0190 PRINT AT 05: "code number"
0200 PRINT AT 0,5: "part name"
0210 PRINT AT 0,5: "quantity"
0220 PRINT AT 0,5: "price"
0230 PRINT
0240 PRINT "Press any key < >..."
0250 wait'for'keystroke
0260
```

After this introduction the program will proceed to the main program loop as soon as the user presses a key. We have used the same **wait'for'keystroke** procedure as in the previous program.

```
0270 REPEAT
0280   show'menu
0290   IF reply$="1" THEN create'record
0300   IF reply$="2" THEN fetch'record
0310 UNTIL reply$="3"
0320
```

The main loop displays a menu then diverts execution to **create'record** or to **fetch'record** in response to a valid user response to the menu:

```
0330 PROC show'menu
0340   PAGE
0350   PRINT ":::::: RANDOM FILE DEMO - MENU ::::::"
0360   PRINT
0370   PRINT
0380   PRINT AT 0,5: "<1> CREATE a record"
0390   PRINT AT 0,5: "<2> FETCH  a record"
0400   PRINT AT 0,5: "<3> terminate"
0410   PRINT
0420   PRINT
0430   wait'for'keystroke
0440 ENDPROC show'menu
0450
```

If response **1** is chosen, the program allows the user to create a record for the inventory file:

```
0460 PROC create'record
0470   PAGE
0480   PRINT ":::::: CREATE A RECORD ::::::"
0490   PRINT
0500   PRINT
0510   INPUT "Which record number: ": no
0520   PRINT
0530   PRINT
```

```
0540   IF no>0 AND no<=maxquantity THEN
0550     INPUT "code number: ": code$
0560     INPUT "part name  : ": part$
0570     INPUT "quantity   : ": quantity$
0580     INPUT "price      : ": price$
0590     OPEN FILE 1,"@0:inventory",RANDOM 128
0600     WRITE FILE 1,no: code$,part$,quantity$,price$
0610     CLOSE
0620   ENDIF
0630 ENDPROC create'record
0640
```

The first part of this procedure is just housekeeping. The user must enter the reference number (1 to 25) of the record to be created. If it is valid, the IF...ENDIF loop is executed. Notice how the random file is OPENed. The first characters in quotes: **0:** indicate that the primary disk drive, drive **0**, is to be used. If a second drive were available and properly connected to the Commodore 64, it could be referenced as drive **2**. (This has to do with Commodore compatibility with the 4000 and 8000 series computers which can have two built-in drives.) The WRITE FILE statement in line 600 transfers the four data elements in the record to the file **inventory**.

A few general remarks on random access files are appropriate here. Data is stored in random files in *binary form*:

* The instruction WRITE FILE causes the data in the record to be saved in binary form on the diskette, where numbers and text take up a certain number of bytes:

integers	take up 2 bytes
real numbers	take up 5 bytes
strings	use up 2 bytes + the string length

The 2 extra bytes for strings are added by the COMAL system to keep track of the string length.

* The Commodore disk drives 1541 and 2031 only allow one RANDOM file to be open at a time.

* In the catalogue of diskette contents, a random access file is classified as a *relative file* and denoted by **rel**.

When we wish to retrieve information which has been stored in the direct file **inventory** the following procedure, activated by user choice **2** from the menu, can be used:

```
0650 PROC fetch'record
0660   PAGE
0670   PRINT "::::: FETCH A RECORD FROM FILE :::::"
0680   PRINT
0690   PRINT
0700   INPUT "Which record number: ": no
0710   PRINT
0720   IF no>0 AND no<maxquantity THEN
0730     OPEN FILE 1,"@0:inventory",RANDOM 128
0740     READ FILE 1,no: code$,part$,quantity$,price$
0750     CLOSE
0760     PRINT
0770     PRINT
0780     PRINT "Inventory item";no;"is:"
0790     PRINT
0800     PRINT "code number: ";code$
0810     PRINT "part name  : ";part$
0820     PRINT "quantity   : ";quantity$
0830     PRINT "price      : ";price$
0840     wait'for'keystroke
0850   ENDIF
0860 ENDPROC fetch' record
```

This procedure requests the user to enter a record number.Then, if a valid record number has been selected, the file is OPENed, and the four data elements of the record are read using the READ FILE statement and printed out.

Suggested improvements:

* This simple program could be improved by adding a counter to keep track of the total number of records in the file. It should be READ as soon as the program is started and updated each time a new record is added or an old one deleted.

* Before the program is used for the first time the file **inventory** should be created in its maximum size. To do this write:

  CREATE "inventory",25,128

  This way you can be sure that there is enough room on the diskette for the complete file. Furthermore, access to the diskette will be substantially faster, because the system need not expand the file as it is used.

* All the records can be zeroed with known data. This can eliminate the possibility of reading undefined records. It also allows the issuing of a warning if useful information is about to be overwritten. One possibility is as follows:

```
OPEN FILE 1: "inventory",RANDOM 128
FOR nr:=1 TO 25 DO WRITE FILE 1:spc$(126)
CLOSE
```

(The inventory is hereby zeroed using blanks. Of course you must be sure there is nothing of value in the file before doing this!)

Moving a Sequential File

The last program in this chapter is intended to illustrate how a sequential file can be transferred from one diskette to another. Files written in machine code are binary files, and moving them can be a problem. The program name is **Move Sequential**, and it is available on your demo diskette or tape.

The key to this program is the statement **GET$(<fileno>,<bytes>)**. By using this statement everything on a diskette, including separators not read in by the INPUT FILE statement, can be read.

The program opens a user selectable sequential file, reads the entire contents into the variable **number$** (however max. 5000 characters), requests the user to switch diskettes and then writes the contents of **number$** to a file with the same name on the new diskette:

```
0010 PAGE
0020 DIM name$ OF 40
0030 INPUT "Enter file name: ":name$
0040 OPEN FILE 2,name$,READ
0050 DIM number$ OF 5000
0060 WHILE NOT EOF(2) DO
0070   number$:+GET$(2,1000)
0080 ENDWHILE
0090 CLOSE FILE 2
0100 PRINT number$
0110 PRINT "Switch diskettes and press any key..."
0120 dummy$=KEY$
0130 WHILE KEY$=CHR$(0) DO NULL
0140 OPEN FILE 3,"@0:"+name$,WRITE
0150 PRINT FILE 3: number$
0160 CLOSE FILE 3
```

File Types

You have noticed that when you view the contents of the diskette using the **dir** instruction that different types of files are stored. At the right next to the file name you will see a three-letter abbreviation describing the file type:

prg	program file
seq	sequential file
rel	relative file
usr	user sequential file

This classification limits the way in which these files can be used. For example if you try to LOAD a relative file as a program, COMAL will generate an error message. Furthermore it enables you to select files from the directory. Try

dir "*prg"

You will probably find it useful as you use files more and more to indicate what the various files within a certain category are used for. You will be working with fonts, shape tables for sprite images, listed sequential files containing programs, procedures or functions, external procedures, display files, textfiles or data files.

To distinguish these files from one another, and to make it possible to show all files of a certain type using the **dir** instruction, it is useful to characterize each file with a *file type code*. You might use a three letter code ahead of or at the end of your file. For example you could indicate that a sequential file consists of a LISTed program as follows:

your program.lst **lst.your program**

A text file from an editor program might be distinguished by using **.txt** at the end of the file name or placing **txt.** at the beginning or **.txt** at the end:

letter.txt **txt.letter**

Prefixes or suffixes indicating file types could be as follows:

.lst	**lst.**	for LISTed files
.dsp	**dsp.**	for DISPLAYed files
.obj	**obj.**	for object code files
.src	**src.**	for source code files
.ext	**ext.**	for EXTERNAL procedures
.bas	**bas.**	for Basic programs
.txt	**txt.**	for text files
.gr0	**gr0.**	for graphics screen files
.gr1	**gr1.**	
.sp0	**sp0.**	for sprite files
.sp1	**sp1.**	
.prc	**prc.**	for procedure files
.fnt	**fnt.**	for fonts

The actual choice is of course up to you, but it can ease communication among COMAL users if the same attribute notation is used. In this connection we recommend using the *prefix*, because this will allow you to catalogue all files of the same type using the DIR instruction.

For example, be sure you have a few text files denoted by the prefix **txt.**, then try the following instruction:

```
dir "txt.*"
```

Only files beginning with **txt.** will be shown.

If the *suffix* convention is used, then an instruction such as:

```
dir "?????.sp?"
```

will only list sprite files with five character file names.

Files and the Screen, Keyboard and Disk Drive

One of the powerful features of the COMAL language file handling system is the ability to communicate with the various input/output devices of your computer. Up to this point we have illustrated communication with the disk drive, but communication with screen, and keyboard is also possible.

In order to direct file operations to a particular device, you should use the unit specifier **unit**. The unit specifier should be followed by one of the following string expressions:

kb:	keyboard
ds:	display screen
lp:	line printer
sp:	serial port
cs:	cassette recorder
u<device>:	device (such as Printer-Plotter)
<drive>:	disk drive number (default 0)

Note that <device> must be a number in the range 0-31, and <drive> is a number in the range 0-15. For example:

```
unit "ds:"
```

will direct COMAL to treat the display as the output device.

It is also possible to reveal the current unit assignment using the special string variable **unit$**. For example:

```
print unit$
if unit$<>"lp:" then
unit "cs:"
```

The first instruction simply prints the current unit. The second sequence will set the default unit to the tape unit, unless the current **unit** is the line printer.

A special feature of the file handling system is the symbol @ which may be the first character of a file name. If it is, then the file will be overwritten if it already exists on the diskette. Note that the drive designation **0:** should also be included to avoid problems with the notorious "save with replace" bug in the C-64 file system (eg. **save "@0:testfile"**).

Using Your Datassette Unit

Although serious file handling really requires the use of a disk drive, Datassette users will be pleased to find that many file operations can be done with a tape unit. Operations with sequential files will, however, be considerably slower than with a disk drive. Random access files cannot be used with a tape unit.

Using the 1520 Printer-Plotter

One of the many useful peripheral devices which you can attach to your Commodore 64 is the 1520 Printer-Plotter. It can be used both for listing programs and results and for drawing graphics images of high quality in up to four colors.

It is quite easy to activate your Printer-Plotter from COMAL. If the Printer-Plotter is properly attached to your serial bus (or to the extra serial bus connection at the rear of the disk drive), you can try the following demonstration. Be sure that the 1520 is turned on. Enter a brief program, then type:

list "u6:

Your program should be listed on the Printer-Plotter.

Other operations with the Printer-Plotter are handled in a similar fashion. Just remember to use the device specification **"u6:**.

You will find a demonstration program **Plotter Demo** on the demo diskette and listed in Appendix H. Try out this program and study the listing to see how to use your 1520 with COMAL.

Review

In this chapter several important topics pertaining to the use of COMAL files have been covered:

* file operations on programs and procedures
* using sequential files for numbers and strings
* using random files
* file types

* using files with input/output devices
* using the 1520 Printer-Plotter.

You should be familiar with the following *concepts* after working through this chapter:

file
storage medium
sequential file
random-access (direct) file
record
data element
bubble sort
file types
device specifications

The following COMAL *instructions* have been discussed:

SAVE - LOAD
LIST - ENTER - MERGE
OPEN FILE - CLOSE FILE
PRINT FILE - INPUT FILE
WRITE FILE - READ FILE
RANDOM
CREATE
GET$

In addition to the examples of the use of files shown in this chapter, you may find it helpful to study details on the formal syntax of these instructions in Chapter 4.

The following *programs* have been discussed in this chapter. They are also to be found on the demo diskette:

Addr List Demo
Random File Demo
Move Sequential
Plotter Demo

The best way to learn about files is to use them to make them work for you. You can use the programs in this chapter as a starting point. Change them and extend them. You will find that mastery of the art of file handling is one of the most valuable skills that you will learn while using your Commodore 64 computer and the COMAL cartridge.

Chapter 7

Peripheral Devices

Introduction

Your Commodore 64 computer is provided with several different means for attaching it to other devices. Compared with other computers in its class there is a generous allocation of input/output connectors included in the base price of the computer:

* IEEE serial bus - for connecting the C64 to disk drive, printers or other devices,
* Datassette tape unit interface,
* Parallel input/output port,
* Cartridge port for connecting games, applications programs or language cartridges like COMAL,
* Control ports (2) for connecting joystick, paddles, etc.

As you can see from Chapter 5 on COMAL Packages it is quite easy to integrate the use of joysticks, paddles or a lightpen into your programs. The use of the IEEE serial bus for communicating with disk drives or printers has been covered in Chapter 6 on COMAL Files. Those who have a Datassette unit are also familiar with its use for saving and retrieving programs and files.

In this chapter we intend to direct our attention to the use of the *RS-232C interface*, *IEEE cartridges*, and particularly to the *parallel port*.

The RS-232C Interface

RS-232C is an industry standard which defines a particular type of serial communication. Data is transmitted as a series of pulses one after the other along a single wire. Figure 7.1 illustrates the transmission pattern which corresponds to the serial ASCII-code for the single letter **C**. This letter has the ASCII decimal code 67, corresponding to the binary number **01000011**.

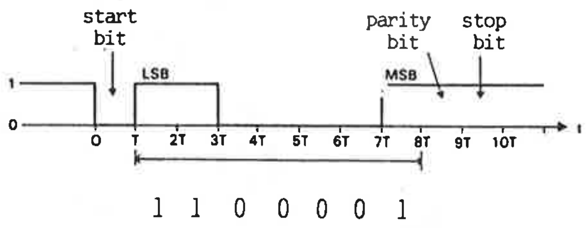


Figure 7.1: The letter C is transmitted in serial form according to the RS-232C standard. Note that only 7 bits are sent, least significant bit first!

The data is sent in *asynchronous* form. The time period for the transmission of a complete character can be divided into 10 equal time intervals. Two well-defined voltage levels determine whether the signal in a given interval is to be interpreted as *high* or *low*. In this discussion we will refer to logic levels, but keep in mind that in practice these levels will appear as voltage variations in the RS-232C connector cable.

Every character signal begins with a *start bit*. It is logic **0** in the example shown in Figure 7.1. The start bit is used to synchronize the receiver with the transmitter. When detected, the start bit starts a clock with period **T** which then coordinates the reading of the serial line. The receiver can take periodic samples to determine whether each bit is a logic **1** or a logic **0**. After seven samples the binary code of the character is available in the receiver's storage register.

The next bit is the *parity bit* which indicates to the receiver whether an even or odd number of **1**'s (or **0**'s) is transmitted in a given character code. For systems with *even parity* the parity bit will be high (logic **1**) if an *even* number of *high* bits are transmitted and low (**0**) if an *odd* number are sent, the parity bit included. This can be checked by the receiver to ascertain whether or not transmission errors have occured. Finally, a *stop bit* is sent to indicate the end of the character transmission.

The RS-232C standard also specifies a *protocol* which is designed to facilitate communication. For example CTS (Clear To Send) and RTS (Request To Send) signals are defined. Furthermore, the voltage levels for logic **1** and logic **0** are specified as -12 and +12 volts respectively. The complete specification can be found in textbooks on electrical engineering. The information which follows should be adequate to allow you to begin using the RS-232C interface with COMAL.

The electrical connections to an RS-232C port are standardized using the DB-25 connector:

pin	signal	code
1	protective ground	GND
2	transmitted data	SOUT
3	received data	SIN
4	equest to send	RTS
5	clear to send	CTS
6	data set ready	DSR
7	signal ground	GND
8	carrier detect	DCD
9-17	... not used ...	
18	ring indicator	RI
19	... not used ...	
20	data terminal ready	DTR
21-25	... not used ...	

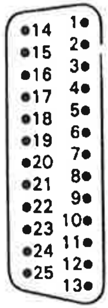


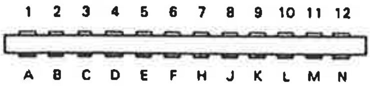
Figure 7.2: The standard pin connections for the RS-232C interface and the pin arrangement for the DB-25 connector are shown.

All available RS-232C control signals are rarely used in actual communications setups. It is often adequate to use only the two data channels SIN and SOUT. An interface of this type is sometimes called a *three line interface* since it consists only of an input, output and ground.

Your Commodore 64 can handle the three line interface as well as the complete RS-232C interface with all control signals. However the Commodore interface deviates from the RS-232C standard with respect to voltage levels. The Commodore 64 uses 0 volts for logic **1** and +5 volts for logic **0**. The RS-232C signals are available on the Commodore user port as indicated in Figure 7.3:

Commodore user port	RS-232C signal description	Signal direction	DB-25 standard connections
A	GND	-	1
B	SIN	input	3
C	SIN	input	3
D	RTC	output	4
E	DTR	output	20
F	RI	input	18
G	DCD	input	8
K	CTS	input	5
L	DSR	input	6
M	SOUT	output	2
N	GND	-	7

(NB: B and C should be connected together)



Commodore 64 user port pin connections.

Figure 7.3: The Commodore RS-232C connections are available on the user port on the rear left-hand side of the computer.

It is very important that the voltage levels of the Commodore 64 RS-232C interface are adapted to the +/- 12 volts present on other equipment. A standard adapter which accomplishes this is available from your Commodore dealer. Diagrams for such devices can also be found in the hobby literature, so that you could build such an interface yourself.

> WARNING: Incorrect connection of the RS-232C interface to other equipment using +/- 12 volts can cause permanent damage to your computer.

Using COMAL you can select a number of parameters to accommodate the requirements of the communications equipment to which your Commodore is connected. The following COMAL program illustrates a way to *receive data* using the RS-232C interface:

```
0010 OPEN FILE 1,"sp:b1200d8s1pe",READ
0020
0030 REPEAT
0040   a$:=GET$(1,1)
0045   PRINT a$,
0050 UNTIL a$=CHR$(255) OR KEY$<>CHR$(0)
0060
0070 CLOSE FILE 1
0080
0090 END "End"
```

Line 10 opens a logical file numbered **1** and specifies the following information: the file opened is to be a file which READs the serial port with a baud rate of 1200 (**b1200**), 8 data bits (**d8**), 1 stop bit (**s1**) and even parity (**pe**). In general the following coding can be used to specify the parameters of the RS-232C interface:

parameter	syntax	range	default
baud rate	b<baud>	50-2400	b300
data bits	d<num>	5-8	d7
stop bits	s<num>	0-2	s2
parity	p<type>	n=none e=even o=odd	pn

Examples:

"sp:"	300 baud, 7 data bits, 2 stop bits, no parity bit
"sp:b600"	600 baud
"sp:b1200d8s1pe"	1200 baud, 8 data bits, 1 stop bit, even parity

Notice that the serial channel will remain open and the program continues to execute the REPEAT-UNTIL loop in lines 30-50 until a transmitted character code corresponds to decimal 255 or any key is pressed.

Data *transmission files* are opened in the same way, using WRITE instead of READ. Notice also that an RS-232C interface file which has been OPENed must of course be CLOSEd again as soon as possible. It is not possible to use the tape recorder or the IEEE serial bus (i.e. the disk drive) while the RS-232C interface is in operation. Thus data which is received must be stored in program memory as it enters the RS-232C port and then saved to disk later.

Similary, you must prepare a data file in working memory, OPEN the RS-232C file, send the data, then CLOSE it before using the disk drive.

File Transfer between Computers

The RS-232C communication channel can among many things be used for file transfer between computers. It doesn't have to be two COMAL computers, but in this section we will show how two COMAL computers communicate. To achieve this it is essential that the computer with which you want to communicate also has an RS-232C input and output connection. Furthermore, because the C-64 RS-232C interface uses TTL-logic levels, you will require a converter module to change 0/5 volt signals to the RS-232C standard levels of -12/+12 volt.

Three-line Interface:

Commodore 64 (C64) and another microcomputer (called PC) are connected using three lines of the DB 25 connector:

C64	PC
pin 2 (in - signal - out)	pin 3
pin 3 (out - signal - in)	2
pin 7 (signal ground)	7
	4
	5
	6
	8
	20

Transfer of COMAL Program Files

The program you want to transmit, is stored in the memory of the transmitting computer (**sender**). Run through the program. If you doubt, that the COMAL of the receiving computer will be able to interpret a program line, then make it a comment line by placing // at the beginning of the line after the line number. In this way you avoid causing the receiver to break the transmission because of a syntax error or losing program lines. By making every doubtful line a comment line, a complete transmission is insured. Later on you can revise the received program by means of the COMAL editing facilities. Notice in this connection that the CHANGE command will be very useful.

Before the transmission, the RS-232 transmission conditions must be specified. On most PC's this is accomplished by means of a configuration program, which sets up the RS-232 communication port. We shall assume, that this is the case. As described earlier in this chapter, these conditions are specified at the start of the transmission on your C64. Make certain that the configurations match on the two computers.

Programs from C64 to PC

Store the COMAL program in the C64 memory and adjust it according to the description of the previous paragraph:

1/ Type on PC: **new** to erase any existing program in memory.

2/ Type on PC:**enter "<name of communication port>"**
The PC is now waiting for data to be received from the communication port.

3/ Type on C64: **select output "sp:b1200d7s1pe"**, followed by **list**. When the RETURN-key is pressed the transmission of data from the C64 serial port is initiated. The transmission form is chosen by the **select output** command to be: 1200 baud, 7 data bits, 1 stop bit and even parity.

4/ The program is now being transmitted from the C64 to the PC. A few syntax errors might show up. If they don't interrupt the transmission, just type in the lines after the transmission has finished. If transmission is interrupted, then make the lines in question comment lines and start transmission all over again.

5/ When the C64 cursor starts flashing the transmission has finished, and the PC waits for a 'terminate'-signal (EOF signal). This signal may differ from computer to computer. Experiment with your set-up. We shall assume that CHR$(26) signals End Of File.

6/ Type again on the C64 (just move the cursor up to the previously typed line): **select output "sp:b1200d7s1pe"**, followed by **print chr$(26)**. The PC cursor ought to start flashing to indicate the end of transmission.

7/ Finally type on the C64: **select output "ds:"** to return output to the display screen.
8/ The transmitted program can now be revised and corrected to fit the PC COMAL.

Programs from PC to C64

The COMAL program, which is to be transmitted, is stored in the PC memory, and if necessary some eventually a few lines are made comment lines to prevent syntax error messages.

1/ Type on C64: **select input "sp:b300d7s1pe"**. Notice the slower transmission rate (remember to adjust the PC configuration accordingly).
2/ Type on PC: **list "<name of communication port>"**
3/ When the PC cursor flashes the transmission has finished. Press **<STOP-RESTORE>** on C64 to interrupt connection to the serial port. The transmited program is now ready for revision.

Sequential ASCII files

It is possible to transmit any sequential ASCII file from one computer to the other. One might mention program files, text files, as well as files with numbers or other sorts of useful information. To accomplish this 4 short programs are needed: a transmitting program and a receiving program for each of the computers. The two C64 programs are always used. But the two PC programs might be adjusted to fit precisely your situation. The difference in programs is due mainly to differences in speed between a 16 bit PC and the 8 bit C64 and the rather slow disk access of your C64.

Normally the following procedure is appropriate:

1/ Make sure that the computers are properly connected via the RC-232C modul at the rear of the C64.
2/ Check that the RS-232 configurations of the two computers match. If transmission fails you might try with a lower transmission rate.
3/ Make sure that the sequential file to be transmitted actually is on the disk of the sender. Load the sender-program to make it ready for execution.
4/ Load the receiver-program into the receiver-computer and **run** the program, answering the question about the file name. The computer now awaits data.
5/ Now **run** the sender-program. Respond to the question about the file name.
6/ Data transmission now begins. Note that 1200 baud equals about 150 characters per second, i.e. about two screen lines per second. Thus a kilobyte takes about 6-7 seconds.
7/ Transmission is completed when the sender transmits a terminate character. You might use CHR$(26) for the PC and CHR$(127) for the C64 End Of Transmission. The receiving program stores the received file on disk.

C64 receiving program:

```
 10 // save "@receive'C64"
 20 DIM a$ OF 1, b$ OF 28000
 30 PAGE
 40 INPUT "Type name of stored file: ": b$
 50 PRINT "Awaiting data..."
 60 OPEN FILE 2,"sp:b600d7s1pe",READ
 70 b$:=""
 80 WHILE a$<>CHR$(127) DO
 90   a$:=GET$(2,1)
100   b$:+A$
110 ENDWHILE
120 OPEN FILE 3,"@"+b$,WRITE
130 FOR i#:=1 TO LEN(b$)-1 DO PRINT FILE 3: b$(i#:i#),
140 CLOSE
150 PRINT "Transmission finished"
```

C64 transmitting program:

```
 10 // save "@transmit'C64"
 20 DIM a$ OF 20
 30 PAGE
 40 PRINT "A file is transmitted from disk to serial port"
 50 INPUT "Type file name: ":a$
 60 OPEN FILE 2, "sp:b1200d7s1pe",WRITE
 70 OPEN FILE 3,a$,READ
 80 PRINT "Transmitting data..."
 90 WHILE NOT EOF(3) DO
100   a$:=GET$(3,1)
110   PRINT FILE 2: a$,
120 ENDWHILE
130 PRINT FILE 2: CHR$(26) // PC EOF (?)
140 CLOSE
150 PRINT "All data transmitted"
```

PC Receiving Program:

```
 10 // receive PC
 20 DIM a$ OF 20
 30 // clear screen
 40 INPUT "Type name of file to be stored on disk: ": a$
 50 OPEN FILE 2,"<name of communication port>",READ
 60 OPEN FILE 3,a$,WRITE
 70 WHILE NOT EOF(2) DO
 80   a$:=GET$(2,1)
 90   PRINT FILE 3: a$,
100 ENDWHILE
110 CLOSE
120 PRINT "All data recieved and stored"
```

PC Transmitting Program:

```
10 // transmit PC
20 DIM a$ OF 20
30 // clear screen
40 INPUT "Type name of file to be transmitted: ": a$
50 OPEN FILE 2,"<name of communication port>",WRITE
60 OPEN FILE 3,a$,READ
70 WHILE NOT EOF(3) DO
80   A$:=GET$(3,1)
90   PRINT FILE 2: a$,
100 ENDWHILE
110 PRINT FILE 2: CHR$(127) // C64 terminate
120 CLOSE
130 PRINT "All data transmitted"
```

IEEE Cartridges

It is possible to purchase a variety of IEEE interface modules which attach to the Commodore 64 cartridge port. Such devices are available from your Commodore dealer (ask for the *IEEE 488 cartridge*) as well as from other suppliers. One of these is called the *Bus Card II* and is available from the company *Batteries Included*. These cartridges can be used with your COMAL language cartridge, for the cartridge bus is accessible in these products. The IEEE cartridge is inserted in the cartridge port, then your COMAL cartridge can then be inserted in a slot in the IEEE cartridge.

The main advantage of the extra IEEE cartridge is that you can then use your Commodore to communicate with high capacity, high speed disk drive units like the Commodore CBM 8050 and 8250 devices.

If you have access to *other cartridges* such as game cartridges, spreadsheets and the like, you must remove your COMAL cartridge in order to use them. In that case be careful to TURN OFF THE POWER to all units in your system before switching cartridges.

The Parallel Port

One of the most useful features of your Commodore 64 is the *parallel input/output port*, the *I/O port* for short. The I/O port can be used to communicate with the outside world. You can use the port as an *output* for control purposes (to run a machine, switch lights on and off, automate an electric train, etc.). The port can also be used as an *input* to gather information (measure voltages, temperatures, and other quantities). In this section we will describe a simple application to illustrate how the port can be used.

This section is not intended to be a complete description of the I/O port. The best place to find details about the parallel port is in the **Commodore 64 Programmer's Reference Guide** available from your Commodore dealer. In the following only as much information as necessary for you to understand the examples will be presented.

The physical location of the port is the edge connector at the far right side of your Commodore 64 when viewed from the rear. The location of the port slot is shown in Figure 1.1 in Chapter 1. Electrical pin connections for the parallel port are shown in Figure 7.3 earlier in the present chapter. To make an electrical hook-up to the port, you will need a 24-pin edge connector plug, available from your dealer or from most electronics supply houses. Note that the connections we will use are as follows:

connection	**signal**
pin 1 (or A)	ground
pin 2	+5 vdc (max 100 mA)
pin C	port B bit 0
pin D	port B bit 1
pin E	port B bit 2
pin F	port B bit 3
pin H	port B bit 4
pin J	port B bit 5
pin K	port B bit 6
pin L	port B bit 7

One convenient way to attach your Commodore 64 to external equipment is by means of a meter long peace of 10 conductor ribbon cable. Solder the 10 leads to the pins of the 24-pin edge connector as indicated above. Solder the other end to a standard DB-25 miniature 25 pin connector. The pin assignments for the DB-25 connector are shown in Figure 7.2. These connectors are quite readily available and inexpensive as they have been adopted as a standard for for the RS-232C interface. Label the connectors carefully. If you make a mistake applying voltages to these connectors, you could damage your computer.

The 25-pin connector is recommended because you may decide to add more connections for advanced projects later on. Use pin 1 on the DB-25 for ground, pin 2 for +5 volts and pins 18-25 for port B bits 0-7.

WARNING: Do not carry out these projects without some prior experience working with electrical connections. Never make connections to the computer unless all power has been turned off. Altough the projects are not difficult, incorrect connections to your Commodore 64 could damage the computer. If you are not sure how to proceed, have an electronically inclined friend give you a hand, or ask your dealer for advice.

To illustrate connection of an external device to the I/O port, we have chosen a simple control project. Once you have understood this example, you should be prepared to tackle more ambitious tasks.

Suppose that we have a closed loop of track, one electric train and a station. We want the computer to allow the train to run around the loop until it approaches the station. It must stop at the station, wait for a predefined period, then run around the loop again.

In order to accomplish this control process, two items of hardware are required:

* A *transistor* and *relay* must be available to switch the power to the train tracks on and off. This is easily accomplished using a few parts readily available from an electronics hobby store.

* A *sensor* must detect the passage of the train just before the station. This can be done using a Darlington phototransistor and a small light source beamed across the track to strike the sensitive area of the phototransistor. The collector should be connected to the port bit as described below, and the emitter should be connected to ground.

Note that in order to control the train, we will need to use two bits of the parallel port. We are free to choose. Let's use **bit 0** for the light detector and **bit 1** for starting and stopping the train.

Each bit of the parallel port B can serve as an input or an output. This is indicated by storing the appropriate number in the *data direction register* for the port, in this case port B. The addresses for the data direction register and for port B are as follows:

	decimal	**hexadecimal**
port B address	56577	$DD01
data direction register	56579	$DD03

The number stored in the data direction register (often abbreviated *ddr*) determines whether the individual bits of port B will act as inputs or outputs. It is easiest to understand the situation using binary numbers. A **0** bit in the ddr means the corresponding bit of port B will act as an input. A **1** bit in the ddr sets the corresponding bit of port B to an output. For example, binary **00000010** (decimal **2**) stored in the ddr will make port B bit 0 an input and bit 1 an output. This is just what we need to control the train.

Because COMAL will accept binary numbers directly, it is not necessary for the programmer to translate the binary number to its decimal equivalent. The programmer must simply remember to preceed binary numbers by the symbol %.

The program **Train Demo** is available on your COMAL demo diskette or tape. It is also listed completely in Appendix H.

Line 10 indicates the name of the file. In line 30 the screen is cleared by PAGE. Lines 40-90 print the following message on the screen:

ELECTRIC TRAIN DEMO

Your train should start at the station with the passage detector just behind the last car. Start the train and then press any key to turn control over to your computer...

Notice line 100:

```
0100 WHILE KEY$=CHR$(0) DO NULL
```

These instructions keep the program in a loop until any key is pressed. The system variable KEY$ will then be different from the null string CHR$(0), and the program will continue.

The main program starts in line 200. The procedure **define'variables** in line 220 defines the addresses of port B and its ddr, and the initial value of the variable **position** is set to **1**. Note the convenient variable names:

```
0680 PROC define'variables
0690   port'b:=$dd01
0700   port'b'ddr:=$dd03
0710   position:=1
0720 ENDPROC define'variables
0730
```

The apostrophes ' are necessary to bind the individual words together, so that COMAL will interpret them as a single variable name, just as with procedure names. The variable **position** will be used to control a pointer on the screen display, indicating the action of the program.

The procedure in line 230 sets port b. This is done as follows:

```
0740 PROC set'port'B
0750   POKE port'b'ddr,2
0760   POKE port'b,2
0770 ENDPROC set'port'B
0780
```

The decimal value **2** corresponds to the binary number **00000010** and makes bit 0 an input and bit 1 an output. Bits 2-7 are not used in this case, so it doesn't matter how these bits in the ddr are set.

The train is started by the procedure **start'train**:

```
0480 PROC start'train
```

```
0490    POKE port'b,PEEK(port'b) BITOR 2
0500    advance'pointer
0510 ENDPROC start'train
0520
```

The POKE instruction places the number **PEEK(port'b) BITOR 2** in the port B address. The BITOR operation is described in detail in Chapter 4. It assures that bit 1 is high. This signal is amplified by the transistor and activates the relay, starting the train. The procedure **advance'pointer** moves an arrow on the screen to the next item of the screen list, jumping back to the start of the list at the beginning of each loop.

```
0790 PROC advance'pointer
0800    PRINT AT 10+position,2: " "
0810    IF position<4 THEN
0820      position:=position+1
0830    ELSE
0840      position:=2
0850    ENDIF
0860    PRINT AT 10+position,2: ">"
0870 ENDPROC advance'pointer
0880
```

The next procedure encountered is the **print'list** procedure. It simply makes a list of items on the computer display:

```
>     train running
      train passes light
      train waiting at station
      Pressing any key will stop the train
      next time it stops at the station...
```

The pointer shows the state of the program.

Now the program enters the main loop:

```
0270 REPEAT
0280    check'light
0290    delay(1.5)
0300    stop'train
0310    delay(10)
0320    start'train
0330 UNTIL KEY$<>CHR$(0)
0340 stop'train
0350 PAGE
0360 END "Au revoir!"
```

This loop will continue to run until any key is pressed. If KEY$ is anything but the null string CHR$(0), the program ends.

The procedure **check'light** examines the state of the bit 0 of port B. This is done as follows:

```
0530 PROC check'light
0540   WHILE PEEK(port'b) BITAND 1 <> 1 DO NULL
0550   advance'pointer
0560 ENDPROC check'light
0570
```

The precise operation of the BITAND operator is described in Chapter 4. In this case the condition **PEEK(port'b) BITAND 1 <> 1** will be FALSE terminating the loop when bit 0 becomes high. This will happen if the light shining on the phototransistor is interrupted. With the collector attached to port B bit 0, the emitter grounded (the base is not used) and the transistor illuminated, the collector-emitter resistance is low (about 100 ohms), pulling bit 0 to low. If the transistor is not illuminated, the resistance becomes high (typically 1 Mohm), and bit 0 returns to the high state.

Before stopping the train, the program executes the procedure **delay(1.5)**:

```
0580 PROC delay(sec)
0590   TIME 0
0600   WHILE TIME<sec*60 DO NULL
0610 ENDPROC delay
0620
```

Note that the variable **sec** is passed to this procedure. It corresponds to the delay time in seconds. TIME resets the internal clock. The loop in line 600 continues until the number of timing units (jiffies = 1/60 sec.) exceeds **sec=60**. Note of course that the parameter value **1.5** can be changed in a particular situation to assure that the train stops as desired at the station.

The train is stopped by the procedure **stop'train** which simply changes bit 1 to the low state. Note that a more refined way to stop (or start) the train would be to rapidly turn the bit off and on, altering the duty-cycle (the proportion of the time the bit is on) gradually from 1 to 0 (or 0 to 1) over a time interval. This will cause the train to gradually slow down (or speed up) in a more realistic fashion. If you decide to do this, replace the relay with a power transistor circuit to control current flow to the track.

The Control Ports

In addition to the many communications possibilities already described, your Commodore 64 computer also has two *control ports* (sometimes called *game ports*). The use of these ports from COMAL has already been described in the section in Chapter 5 on COMAL packages.

In addition to 2 x 5 switch inputs (JOYA0-3, JOYB0-3, BUTTON A and BUTTON B) available at the two control ports, a total of 4 different analogue inputs are also available via the game ports. These inputs are POTAX, POTAY, POTBX and POTBY. (Internally the SID has just 2 ADC's and an analogue switch.) Pinouts and connections are as follows:

pin	**game port A**	**game port B**
1	JOYA0	JOYB0
2	JOYA1	JOYB1
3	JOYA2	JOYB2
4	JOYA3	JOYB3
5	POTAY	POTBY
6	BUTTON A	BUTTON B
7	+ 5V	+ 5V
8	GROUND	GROUND
9	POTAX	POTBX

Note: Maximum load on the + 5V supply is 50 mA.

Note that you will need a standard DB-9 *female* connector to attach experiments to the game ports.

The switch inputs can indicate to a program whether a given switch is **on** or **off**. Examples of how to use these signals are available in Chapter 5.

The analogue inputs go to *A/D converters* which are used to digitize the positions of potentiometers on paddles. The conversion process is based on the time constant of a capacitor tied from the POT pin to ground, charged via a potentiometer tied from the POT pin to +5 volts. The component values may be estimated from the relation: **RC = 4.7E-4**. In this equation **R** is the maximum resistance of the potentiometer and **C** is the capacitance. The larger the capacitor, the lower the uncertainty in the POT value. The recommended values for **R** and **C** are 470 kiloohm and 1000 pF. Note that a separate potentiometer and capacitor are required for each POT pin.

Although the POT inputs in the game ports were designed to measure the rotational position of a potentiometer, any variable resistance can be used. For example to measure temperature simply replace the potentiometer with a thermistor in the proper resistance range. Other resistive sensing devices can of course be used to allow automated recording of pressure, liquid level, illumination or other physical quantities. For example the following program illustrates how you might construct a simple *digital thermometer* using the game port inputs:

```
0010 // save "@0:Thermometer"
0020 USE paddles
0030 // capacitor: 1000 pF
0040 // thermistor: 100 K at 20 degrees
0050 a:=1; b:=0
0060 PAGE
0070 PRINT "DIGITAL THERMOMETER"
```

```
0080 PRINT AT 5,1: "Thermistor and capacitor must be con-"
0090 PRINT "nected to controlport 1..."
0100
0110 // Main program
0120 REPEAT
0130   check'paddle(1)
0140   convert(average)
0150   print'temperature
0160 UNTIL KEY$<>CHR$(0)
0170 END // Main program
0180
0190
0200 PROC check'paddle(port)
0210   total:=0
0220   FOR i:=1 TO 50 DO
0230     paddle(port,a'paddle,b'paddle,a'button,b'button)
0240     total:=total+a'paddle
0250   ENDFOR i
0260   average:=total/50
0270 ENDPROC check'paddle
0280
0290 PROC convert(average)
0300   temp:=a*average+b
0310 ENDPROC convert
0320
0330 PROC print'temperature
0340   temp:=INT(temp*10)/10
0350   PRINT AT 10,10: "T =       o"
0360   PRINT AT 10,14: temp
0370 ENDPROC print'temperature
0380
```

The first part of the program (lines 10-100) are just introductory information, a display message and definition of the constants **a** and **b**. Notice that these are set equal to **1** and **0** respectively in line 50. This causes the program to just printout ADC values (0-255) with no conversion to temperature. These values can first be found after you have constructed a test circuit and *calibrated* the sensor which you plan to use.

Notice the structure of the rest of the program. The main program is from line 110 through line 170. It consists of a REPEAT-UNTIL loop which will be terminated if any key is pressed. In the loop information is fetched from the paddle port by the procedure **check'paddle(1)**. Then this quantity is converted to a temperature value using the procedure **convert(average)**. Finally the procedure **print'temperature** displays the computed temperature on the display screen.

Make a trial setup using a 1000 pF capacitor and a thermistor (NTC or PTC resistor) with a room temperature value of about 100 kohm. Connect your test circuit to the control port as shown in the following figure:

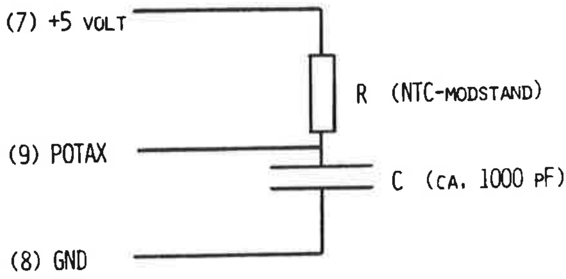


Figure 7.4: Many different sensor types can be attached to the control ports. You can make use of up to 4 analog inputs to the two control ports.

If the program is now run, the measured ADC values will be shown on the screen. Draw a graph displaying the temperature in degrees as a function of the ADC values. If the graph is approximately *linear* in the region of interest, you can compute the constants **a** and **b** as follows:

Read two coordinate pairs from your graph (X1,Y1) and X2,Y2). (X1 and X2 correspond to ADC values, and Y1 and Y2 correspond to temperatures.) The constant **a** can now be found using the formula:

a = (Y2 - Y1)/(X2 - X1)

This is the *slope* of the line you drew on your graph. We found the following values in our test setup which used an NTC resistor: (183,25) and (215,20) - temperatures are in degrees centigrade. I.e. the program showed 183 as ADC value when the sensor temperature was 25 degrees C, and 215 when the temperature was 20 degrees. Thus **a** equals -0.178 in this case. To find **b** you can now used the equation: **temp = a*average + b** (used in the procedure **convert**). Inserting **average** = 183 and **temp** = 25 into this equation yields a value for **b** of **57.6**. If you change line 50 to reflect the new values you have found for **a** and **b**, the program should print out the temperature when you run it again.

If you want to calibrate a sensor over a wider range of temperature, you can use e.g. an exponential function to achieve a better calinbration than the linear approximation we have used in the illustration above. In this case you must revise program linie 300.

Review

In this chapter we have considered a range of possibilities for the use of the wealth of interfacing facilities available with your Commodore 64 computer. You are encouraged to experiment with the RS-232 interface, the parallel port and the game ports to learn more about them.

You will find more information about these ports in the **Commodore 64 Programmer's Reference Guide.** A great deal additional information is also available from the popular literature about microcomputers.

Chapter 8

COMAL and Machine Language

What is machine language?

The "brain" in every microcomputer is a central microprocessor. Your Commodore 64 is no exception. There are a number of different types of microprocessors available, each with its own set of instructions. The Commodore 64 uses a more advanced version of the 6502, the 6510. It uses the same instruction set as the popular 6502 but has additional built-in I/O facilities. The only language a microprocessor can interpret directly is machine language. Any higher level language must ultimately communicate with the microprocessor using its native language.

Inside your COMAL cartridge are a large number of machine code routines termed collectively *the COMAL system*. When the computer is turned on, the COMAL system automatically takes charge of the Commodore 64. Another important machine code program in your computer is *the operating system* which takes care of communication with the keyboard, screen editing and other housekeeping chores. When a COMAL program is "run", appropriate machine code routines are brought into play to achieve the actions which your COMAL statements require.

It should be made clear at the outset, that this chapter is not intended to serve as a tutorial in machine language programming. We assume here prior knowledge of 6502 machine language programming. The material presented here is substantially more difficult than the material in previous chapters. If you want to learn more about 6502 machine language, a number of excellent books are available. You might want to begin with Ian Sinclair's **Introducing Commodore 64 Machine Code** (Granada Publishing, London, 1984). **The Programmer's Reference Guide** available from your Commodore dealer is also a valuable resource.

For further information on how COMAL actually works on the Commodore 64, read **Jesse Knight: COMAL 2.0 PACKAGES** available from COMAL Users Group U.S.A.

Machine language will probably be easier to learn if you can share the learning experience with others who have similar interests. In this connection the many Commodore 64 and Commodore COMAL users groups can provide useful opportunities of exchange of information. Here are some addresses which may be helpful:

In the USA:
COMAL USERS' GROUP, 5501 Groveland Terrace, Madison WI 53716

In Canada:
TPUG Inc., COMAL USERS' GROUP, 1912-A Avenue Rd., Ste.#1 Toronto, ONT M5M 4A1, CANADA

In England:
ICPUG, ATT: Brian Grainger, 73 Minehead Way, Stevenage, Herts SG1 2HZ, ENGLAND

In this chapter you will find an overview of the use of computer memory by the COMAL system. Next comes step by step instructions showing how you can incorporate your own machine code routines as a package in a COMAL program.

Machine language is much easier to work with, if you have access to a 6502 *assembler program.* Such a program allows you to prepare a program using *symbolic machine code* using mnemonic codes instead of programming directly in hexadecimal notation. A disk drive will also make working with machine language easier. On the demo diskette (or cassette) you will find a textfile with the name **C64SYMB**. It contains a list of all instructions which are relevant when doing machine language programming with your Commodore 64 and COMAL. This textfile should be included in the assembler source code with COMAL packages.

It is also possible to prepare a machine language program directly in memory from a COMAL program by using POKE orders. In this way a machine code program can be stored in an available area of computer memory then started from a COMAL program by using **SYS <start address>**. The last instruction in the machine code routine should be an RTS, which causes program execution to return to COMAL. It is, however, not possible using this method to prepare machine code program packages which can be LINK'ed to COMAL programs. In this chapter we will only treat the preparation of machine code programs which can be LINK'ed to a COMAL program.

The use of machine code routines is an integral part of the COMAL system. When designing machine code facilities, three primary goals have been strived for:

* Machine code routines should be easy to use - also for users without knowledge of machine code.
* Access to machine code routines should be by name, thereby eliminating confusing details like memory addresses.
* Machine code routines should be affected by commands like NEW and RUN. In this way packages behave as if they are an integral part of the COMAL system.

There are three commands/statements in COMAL which are used in connection with the definition, use and removal of machine coded routines:

LINK <filename>	// Enter a module file
USE <package>	// Define procedures
DISCARD	// Remove all modules

These commands (USE can also be used as a program statement) will be explained in detail. Machine code routines use the procedure and function mechanism in COMAL and allow therefore all parameter types.

Modules

The LINK command fetches a machine language module (object file) from the library which has been prepared by the assembler. This module contains information which specifies where the machine code is to be located in memory. COMAL can control up to 10 such modules at any one time. At least 2 modules, containing the following, are always defined:

(Module 1)	**(Module 2)**	
english	**graphics**	**sound**
dansk	turtle	joysticks
system	sprites	paddles
	font	lightpen

These modules need not be LINK'ed, for they are already available in the COMAL cartridge. Modules can be removed again using the DISCARD command. However the above mentioned standard modules can NOT be removed. Because the modules are not named, all other modules will always be deleted by DISCARD. Modules can be made permanent (be ROM'ed), whereby they can not be DISCARD'ed. Non-permanent modules are treated as if they were part of the program in working memory. A SAVE order will store all non-permanent modules with the COMAL program in the same **prg** file. When LOAD, RUN or CHAIN is used, they will be read in again (be LINKed).

Packages

A module can contain 0, 1 or more *packages*.

Procedures and Functions

A package can contain 0, 1 or more procedures or functions. Two main elements constitute each procedure or function:

* A *procedure header*, which specifies how many and what type of parameters are to be passed to the procedure.
* The *procedure body*, i.e. the machine code which is to be executed when the procedure is called.

This drawing illustrates the hierarchical structure:

```
- module 1 -  - module 2 -  - module 3 -      -...       module n
     .             .             |
     .             .             |             .
     .                           |
- package 1 - - package 2 -     - ....    - package m -      ...
     .             |              .
     .             |                          .
     .             |
 - proc 1 -     - proc 2 -     - proc 3 -    ..proc p..
     |             |              |             |
  header 1      header 2       header 3      header p
  body 1        body 2         body 3        body p
```

The USE statement performs the following actions: Each module, starting with the last one to be read in, is checked to see if the name following USE is to be found in the list of package names in this module. If the name is found, then the procedures and functions found in this package are defined. The locations of the procedure headers are noted.

Signals

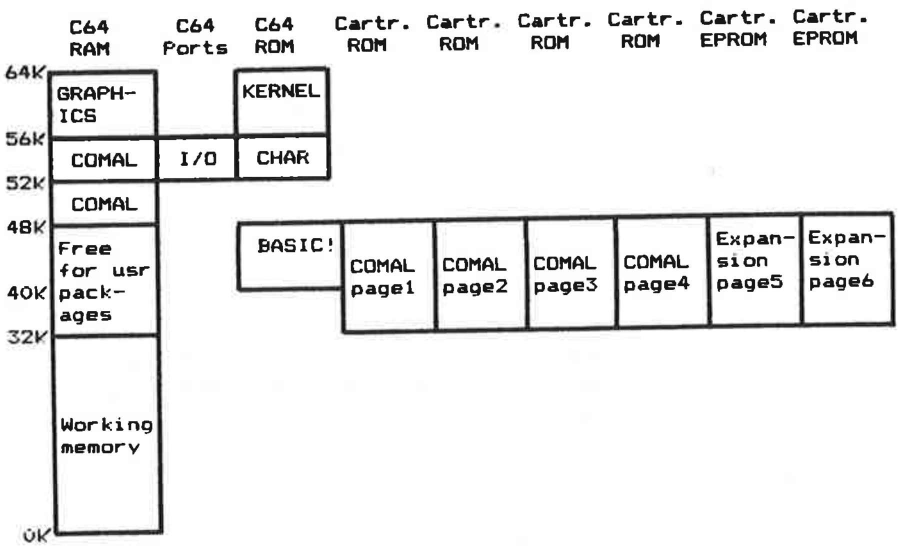
When COMAL carries out an operation which can affect modules or packages, a signal is issued regarding the operation in question. The module or package may or may not react to the signal. There are two types of signals:

* A signal is sent to a package when a USE statement is encountered which activates the package. The signal is in effect a call to a routine which is local for the package. As an example of what such routines may do, the TURTLE package selects the SPLITSCREEN display, when the command USE turtle is given. The main purpose of the routine is to initialize the variables in the package.
* On system start or when LINK, LOAD, DISCARD, NEW, RUN, CHAIN are issued (and in certain other special situations), signals are sent to all modules. The signal causes a call to a routine in the module (and thus common to all packages in the module). The purpose of the signal call is to integrate all packages in the module into the COMAL system (after start-up, LINK, LOAD), or to return the COMAL system to its original state (after DISCARD, NEW). If a package is to use in-

terrupt (IRQ), then the module can link the interrupt routine using LINK and disconnect it again with DISCARD.

How is memory organized?

The following diagram illustrates the entire memory of your Commodore 64 (the first 3 columns), the memory in the COMAL cartridge (the next 4 columns), and finally the user-programmable EPROM expansion (the last 2 columns). The expansion option consists of an empty EPROM socket in the COMAL cartridge. This cartridge can hold an 8KB-, 16KB-, or 32KB-EPROM.



RAM is partitioned as follows:

0- 1KB	System variables for KERNEL, COMAL, processor stack.
1- 2KB	Screen memory.
2-32KB	Storage for COMAL program, name table and stack. Here is also room for packages, which take up user memory. The character set, if used, is at 27-32KB.
32-48KB	Is unused. Packages can be placed here without reducing available program working memory.
48-52KB	COMAL system variables, variables for standard packages.
52-56KB	Variables for function keys, moving sprites, sprite drawings and color information for graphics.
56-64KB	Graphics bit map.

The I/O area contains the input/output ports. All communication with the surrounding world is carried out via these ports. The color memory for the text screen is also located here. This color memory is (unfortunately) shared with multi-color graphics.

The following ROM areas are located in the C-64:

40-48KB	BASIC interpreter
52-56KB	Standard double character set (font)
56-64KB	KERNEL. This is the Commodore 64's operating system. It contains among other things routines for communicationwith the screen, cassette tape, disk drives and the RS232interface.

The **COMAL cartridge** is partitioned into four pages,each containing 16 KB. The are all located in the adressrange 32-48KB. In this way the 64KB COMAL interpreter only takes up 16KB in your Commodore 64.

The contents of the cartridge ROM's are as follows:

Page 1	COMAL starts here when the machine is turned on. It contains the math routines, commands and the packages ENGLISH, DANSK and SYSTEM.
Page 2	The COMAL editor, syntax analysis and code generation, prepass (SCAN), recreator (LIST) commands.
Page 3	Runtime-module.
Page 4	The packages GRAPHICS, TURTLE, SPRITES, FONT, SOUND, JOYSTICKS, PADDLES and LIGHTPEN are located here.

EPROM expansion in the cartridge is interpreted as follows, depending on EPROM type:

8KB	Page 5, address area **$8000-$9fff.**
16KB	Page 5, address area **$8000-$bfff.**
32KB	Page 5, address area **$8000-$bfff,** Page 6, address area **$8000-$bfff.**

Upon start-up COMAL examines every 4 KB in pages 5 and 6 to find certain bytes which determine if package modules are present. Next, signals are sent to the modules, indicating that the machine has been turned on.

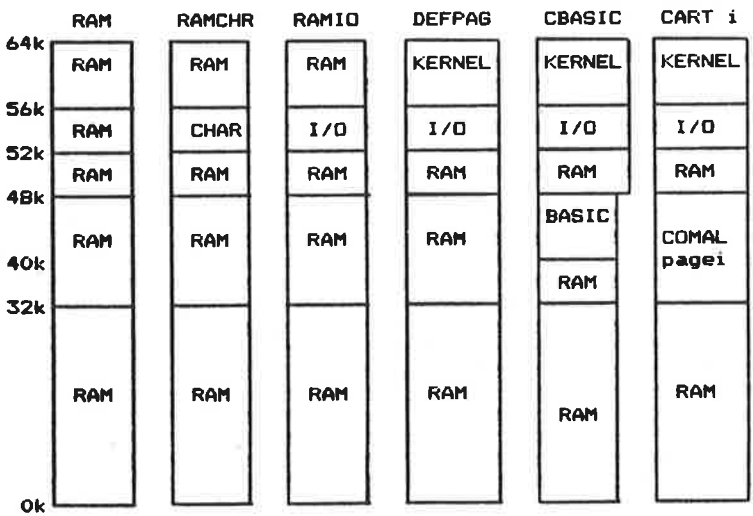
Memory Management

The 6510/6502-microprocessor which is used by the Commodore 64 is not designed to address more than 64 KB. When the COMAL cartridge is active, the processor can address up to 152 KB! A special trick has to be used to achieve this. The trick is to determine just what the 6510 should be able to "see" in its address space. Memory is partitioned into banks (also called pages or overlays). The different banks become active as required. The method is termed "bank-switching" or "memory management". For example there are three banks in the address space 52-56KB: RAM, I/O and character set ROM (see memory manager organization). In the region 40-48KB there are actually 8 different banks which can be used!

Banks are selected by writing a bit pattern into certain control ports. Two such control ports are available:

R6510	Controls the C-64 memory map. Located in the Commodore, address $0001. Can be written to or read.
OVRLAY	Control of cartridge banks. Located in the COMAL cartridge at address $de00 in bank I/O. I.e. the port must be accessible when it is to be changed. It can only be written to.

COMAL has system routines, which manipulate these ports. By using these routines, one can specify the **memory map** by simply altering a single byte. The following figure specifies several interesting memory maps (i=1,2,..,6):



Creating Modules

In order for LINK, USE and DISCARD to work, the placement of code and the format for package names, procedures and procedure headers must be specified.

If a module is to be placed in RAM, then it must have the following format:

```
     .lib c64symb
     *=<start address>
     .byte <map>
     .word end
     .word <signal>
     <package table>
     <machine code>
end    .end
```

If the module is to be placed in EPROM, then is must beformatted as follows:

```
     .lib c64symb
     *=3<start adress>
     .word cold
     .word warm
     .byte 'CBM80comal
     .byte >3*
     .byte <map>+rommed
     .word end
     .word <signal>
     <package table>
     <machine code>
end    .end
```

.lib c64symb makes all KERNEL- and COMAL variables known to the module.

<start adress> is the starting adress for the module in memory.

<map> indicates into which memory map the module is to be placed by LINK. This memory map is automatically activated by calling a procedure, function or signal handler in the module.

rommed indicates that the module cannot be DISCARD'ed.

end is the end address of the module + 1.

<signal> is the signal handler for the module, located in **<machine code>**.

<package table> is a list of package names.

<machine code> is all other code in the module.

A <package table> has the following format:

```
.byte l1,'package1'
.word proct1,init1
.byte l2,'package2'
.word proct2,init2
.
.
.byte 0          ;End of the table
```

li	is the number of characters in the i'th package name.
'packagei'	is the name of the i'th package (in quotation marks).
procti	is the address of the table of procedure names for the i'th package.
initi	is the address of the initialization routine for the i'th package.

A *table of procedure names* must have the following format:

```
procti    .byte l1,'proc1'
          .word proch1
          .byte l2,'proc2'
          .word proch2
          .
          .
          .byte 0     ;End of the table
```

```
procti    .byte l1,'proc1'
          .word proch1
          .byte l2,'proc2'
          .word proch2
          .
          .
          .byte 0     ;End of the table
```

lj is the number of characters in the j'th procedure name.
'procj' is the name of the j'th procedure (in quotation marks).
prochj is the address of the j'th procedure header.

A procedure header has this format:

```
prochj     .byte proc,<codeh,>codeh,n
           .byte  <parameter1>
           .byte  <parameter2>
           .
           .
           .byte  <parametern>
           .byte endprc
```

A function header has this format:

```
funchj     .byte func+type,<codeh,>codeh,n
           .byte  <parameter1>
           .byte  <parameter2>
           .
           .
           .byte  <parametern>
           .byte endfnc
```

type	is the function type (real, int or str).
codeh	is the address of the assembler code routine.
n	is the number of formal parameters.
<parameterk>	is the specification of the k'th parameter.

A parameter specification is one of the following:

```
.byte    value+type               ;Simple value parameter
.byte    value+array+type,dim     ;Array value parameter
.byte    ref+type                 ;Simple reference parameter
.byte    ref+array+type,dim       ;Array reference parameter
```

type	is the parameter type (real, int or str).
dim	is the dimension of an array parameter.
real	means the type is REAL.
int	means integer type (INTeger).
str	means the string type (STRing).

An example of how a procedure header is coded:

FUNC pip(x,y#,REF z$(,)) can be coded as

```
.byte func=real,<pip,>pip,3      ;Real func. with 3 param.
.byte   value=real               ;x
.byte   value=int                ;y#
.byte   ref=array=str,2          ;REF z$(,)
.byte endfnc                     ;No more parameters
```

Parameter Passing

When the COMAL interpreter passes control to an assembler coded routine, all actual parameters (if any) are computed. At the same time parameter types are checked for agreement with the procedure header specification. The number of parameters in the procedure call must also be correct.

It is not possible to know in advance where the parameter value or the variable (when using REF) are located in storage. Therefore it is necessary to call a system routine FNDPAR (FiND PARameter) to obtain information about the storage address of a parameter. Then the parameter can be handled.

FNDPAR: When called: .A is the number of the parameter.
On return: COPY1 contains parameter address.
All registers are changed.

Note: In the COMAL system the following conventions apply:integers and real numbers are stored in high/low format, while addresses are saved in low/high format. This is trueof actual parameters, also for parameters of systemroutines.

In the following the format for each parametre type is described:

VALUE+REAL and REF+REAL

(COPY1)+0:	**5 bytes floating point**	**Exponent+128**
+1:		**Mantissa(1)**
+2:		**Mantissa(2)**
+3:		**Mantissa(3)**
+4:		**Mantissa(4)**

VALUE+INT and REF+INT

(COPY1)+0:	**2 bytes integer**	**High byte**
+1:		**Low byte**

VALUE+STR and REF+STR

m: Maximum string length (dimensioned length).
n: Actual length (If VALUE+STR, then m=n.)

(COPY1)+0:	**m**	**High byte**
		Low byte
+2:	**n**	**High byte**
		Low byte
+4:	**m bytes**	**s$(1:1)**
		s$(2:2)
		s$(3:3)
		s$(4:4)
+4+n-1:		**s$(n:n)(last char.)**
+4+m-1:		**s$(m:m)**

VALUE+ARRAY+REAL, VALUE+ARRAY+INT, VALUE+ARRAY+STR, REF+ARRAY+REAL, REF+ARRAY+INT, REF+ARRAY+STR

Every array has an information block:

n : Number of indices.
addr: Address of first element in the table.

(COPY1)+0:	**addr**	**Low byte**
		High byte
+2:	**n**	**Number of indices**
+3:	**Lower limit for 1st index**	**High byte**
		Low byte
+5:	**Upper limit for 1st index**	**High byte**
		Low byte
+7:	**Lower limit for 2nd index**	**High byte**
		Low byte
+9:		
+3+(n-1)*4+2:	**Upper limit for n'th ind.**	**High byte**
		Low byte
+3+n*4:		

If an array A is declared as:

DIM a(1:3,6:8)

it is placed in memory as follows:

```
addr +0 :a(1,6)
     +1  :a(1,7)
     +2*l:a(1,8)
     +3*l:a(2,6)
     +4*l:a(2,7)
     +5*l:a(2,8)
     +6*l:a(3,6)
     +7*l:a(3,7)
     +8*l:a(3,8)
```

where l is the size (in bytes) of each array element. Each element is organized just as a simple parameter.

Where can modules be placed?

Modules can be placed in RAM from **$0900-$7fff** and from **$8009-$bfff**.

In addition packages can be placed in an EPROM in the cartridge from **$8000-$bfff**, however the start address must be a multiple of $1000.

Where can the module variables be placed?

Variables which much survive from call to call must be placed in the module itself (for RAM-modules).

EPROM-module variables can be stored from $c855-$c87a.

Should more storage be required, and if the RS232 will not be used, then the RS232 buffer RSOBUF (256 bytes) can be used. If cassette tape will not be used, then the tape buffer TBUFFR (192 bytes) can be used. In addition zero-page locations $4c, $56 and $fb-$ff can be used freely.

Routines which use variables local to the individual call can use these local variables:

Name	**Address**
INF1	$0038
INF2	$0039
INF3	$003a
Q1	$003b-$003c
Q2	$003d-$003e
Q3	$003f-$0040
Q4	$0041-$0042
Q5	$0043-$0044

COPY1	$0045-$0046	Also used by FNDPAR
COPY3	$0047-$0048	
COPY3	$0049-$0050	
AC1	$0061-$0066	Also used by FP-routines
AC2	$0069-$006f	
MOVEAD	$007a-$007b	
TXTLO	$007c	
TXTHI	$007d	
RANGES	$02e0-$02ff	
TXT	$c760-$cyaf	

Signal Routines

A signal routine is a subroutine which is terminatedby an RTS instruction. It s permissible for a signalroutine to do anything which a procedure or a function may do. If a signal routine is not required, then a systemroutine named DUMMY can be used. This routine consists ofonly an RTS instruction and does nothing.

A **USE-signal-routine** has no parameters. Each time a USE<**package**> statement is encountered in a COMAL program,this routine is called. If it is not desired that the package be initialized every time, then a variable should by used toindicate that a package has previously been activaed by means of USE

A **module-signal-routine** has one parameter, for the .y-register will contain a value when the call is executed,indicating which type signal is to be transmitted. Theparameter can be one of the following:

POWER1

Is issued at start-up to all ROM'ed modules. The signal must be used to initialize the module.

POWER2

Is issued at start-up after POWER1 has been issued. Ordinarily this signal is ignored, but it can be used to allow a module to take complete control before COMAL starts.

LINK

Is issued to a just LINK'ed package or to those packages which are read in with LOAD, RUN <**filename**>, or CHAIN. With this signal the module can change vectors in COMAL and the operating system.

DSCRD

Is issued to all modules before DISCARD or the NEW command. On this signal the module can change vectors back to what they were before LINK.

NEW
Is issued with a NEW command.

CLRTAB
Is issued when all names in a program are undeclared. This signal is given with the RUN and CHAIN commands and in certain other cases. When the names are undeclared, then it is not possible to call any procedure or function in any package.

RUN
Is issued with the RUN or CHAIN command.

WARM1
Is issued during "warm start", i.e. when the <**STOP-RESTORE**> combination is activated from the keyboard.

CON
Is issued with the CON command.

ERROR
Is issued after the program has stopped with an error message.

STOP1
Is issued after a program has stopped due to a STOP or END.

BASIC
Is issued before COMAL is exited.

In general a module-signal-routine follows this outline:

```
signal      cpy #link       ;LINK-command?
            beq slink       ;Jump if so
            cpy #dscrd      ;DISCARD?
            beq sdscrd      ;Jump if so
            rts             ;Ignore all other signals.
;
slink       ..              ;LINK-handler
            rts             ;Back to COMAL

sdscrd      ..              ;DISCARD-handler
            rts             ;Back to COMAL
```

Error Reporting

It is good programming practice to check whether parameters to a procedure or function are legal. It they are not, then an error message should be issued. If it is desired that COMAL's own error messages be used, this can be done as follows:

```
ldx #5          ;Give error number 5
jmp runerr      ;i.e. "value out of range"
```

With this method one can give standard error messages numbered 0 to 255. See Appendix F for these error messages. RUNERR corresponds closely to the COMAL statement REPORT <**error**> and can be captured in a TRAP structure, if this is desirable.

A more general *error reporting method* is available. If one wants to give the following values to the system or to an error handler,

```
ERR          = 300
ERRFILE      =   0
ERRTEXT$     = "illegal parameter value"
```

it can be done with the following routine:

```
text      .byte 'illegal parameter value'
          textl=*-text
;
err300    ldx #textl          ;Length of text
          stx ertlen          ;Length of error message
errorp    lda text-1,x        ;Move the text to ERTEXT
          sta ertext-1,x
          dex
          bne errorp
;
          lda #$6c            ;Copy jmp (trapvc) to Q1
          sta q1+0
          lda #<trapvc
          sta q1+1
;
          ldy #0              ;ERRFILE = 0
          ldx #<300           ;ERR = 300
          lda #>300
          jsr goto            ;Execute jmp (trapvc) in PAGEB
          .byte pageb,<q1,>q1
```

Package example

The following example shows how a complete modulecontaining one package named **TEST** can be created.The purpose of this example is to illustrate how onecreates a procedure, a **real** function and a **stringfunc-**

tion. The package is placed from address **$8009** in RAM in the memory map DEFPAG (see the table of useful memory maps shown earlier).

The package is available on the demo diskette.

test.src	contains the source code (src=source).
test.obj	contains the object code (obj=object).

In order to get the module with the package **test** into the machine, type:

```
LINK "test.obj"
```

Next type in:

```
AUTO
0010 USE test // makes hi, add and string known //
0020 hi
0030 PRINT add(23,45)
0040 PRINT string$("a",10)
0050            (Press the <STOP> key.)
```

RUN which gives this result:

```
hello!
68
aaaaaaaaaa

end at 0040
```

Switch to your own diskette then type:

SAVE "test"	save the COMAL program and the package test.
DISCARD	delete the LINK'ed module.
RUN	run the program again without the package **test**.

The system will respond with an error message:
at 0010: test: unknown package

RUN "test" fetch and run the program with the package **test**.

New printout:

```
hello!
68
aaaaaaaaaa

end at 0040
```

Here is the content of the source code of **test.src**:

```
;           ---=== package test===---
;
;           make all symbols known:
;
            .lib c64symb
            .opt list                  ;list this module
;
            *=$8009                    ;start address
;
            .byte defpag               ;52KB RAM memory map
            .word end                  ;the module ends with end
            .word dummy                ;no signal handler
;
;package table:
;
            .byte 4,'test'             ;the package is called test
            .word testp                ;procedure table
            .word dummy                ;no initialization
            .byte 0                    ;no more packages
;
;procedure table:
;
testp       .byte 2,'hi'               ;the procedure hi
            .word phi                  ;procedure header for hi
            .byte 3,'add'              ;the function add
            .word padd
            .byte 6,'string' ;function string
            .word pstrin
            .byte 0                    ;no more procedures
;
; proc hi
;
phi         ,byte proc,<hi,>hi,0       ;no parameters
                                       ;begins in hi
            .byte endprc

;func add(a#,b#)
;
padd        .byte func+real,<add,>add,2 ;two parameters
                                       ;begins in add
            .byte value+int            ;a# is integer value parameter
            .byte value+int            ;b# is integer value parameter
            .byte endfnc
;
```

```
;func string$(character$,number#)
;
pstrin    .byte func+str,<string,>string,2;two parameters
                                          ;begins in string
          .byte value+str                 ;character$ is string value parameter
          .byte value+int                 ;number# is integer value parameter
                                          .byte endfnc
;
;
;proc hi
;         print "hello!"
;endproc hi
;
text      .byte 'hello!',13               ;text to be printed
textl     =*-text                         ;length of text
;
hi        ldy #0                          ;begin with 1. character
hilp      lda text,y                      ;fetch character
          jsr cwrt                        ;print character on screen
          iny                             ;next character
          cpy #textl                      ;finished?
          bne hilp                        ;jump if not finished
          rts                             ;return to COMAL
;
;
;func add(a#,b#)
;return a#+b#
;endfunc add
;
add       lda #1                          ;get address of 1. parameter
          jsr fndpar                      ;copy1 = address
          ldx copy1                       ;move address to copy2
          lda copy1+1
          stx copy2
          sta copy2+1

;
          lda #2                          ;get address of 2. parameter
          jsr fndpar
;
;copy1 points now to b# and copy2 points now to a#
;
          ldy #1                          ;NB: integers are in high/low format
          clc                             ;no carry
          lda (copy2),y                   ;low byte of a#
          adc (copy1),y                   ;plus low byte of b#
```

```
        tax                     ;.a is moved over to .x
        dey                     ;.y:=0
        lda (copy2),y           ;high byte of a#
        adc (copy1),y           ;plus high byte of b# plus carry
        bvs ovrflw              ;jump if arithmetic overflow
;
;       .x = low byte of a#+b#
;       .a = high byte of a#+b#
;
;       convert from integer to real number;
        then put result on COMAL's stack.
;
        jsr pshint              ;convert and push
        rts                     ;return to COMAL with the result
;
ovrflw  ldx #2                  ;"overflow"
        jmp runerr              ;report 2
;
;
;func string#(character$,length#) closed
;       if length#<0 then report 1 // argument error //
;       if len(character$)<>1 then report 1 // argument error //
;       dim r$ of length# // room for result //
;       for i#=1 to length# do  // generate result //
;       r$:+character$
;       endfor i#
;       return r$ // return result //
;endfunc string
;
num     =copy2                  ;use copy2 as num
;
string  lda #2                  ;get address of 2. parameter
        jsr fndpar
        ldy #0                  ;test sign
        lda (copy1),y
        bmi argerr              ;jump, if <0
        sta num+1               ;high byte of num
        iny                     ;.y:=1
        lda (copy1),y
        sta num                 ;low byte of num
;
;generate the result directly on COMAL's evaluation stack.
;
;stos points to the next free byte on the stack
;the stack is limited upwards by sfree
;test if there is room for the result
```

```
;
        clc                     ;clear the carry
        adc stos                ;num+stos
        tax                     ;.x:=low byte of num+stos
        lda num+1
        adc stos+1              ;.a:=high byte of num+stos
        bcs sterr               ;jump, if overflow
;
        tay
        txa                     ;num+stos+2
        adc #<2                 ;the carry is known to be = 0
        tax
        tya
        adc #>2
        bcs sterr               ;jump, if overflow
;
        cpx sfree               ;if num+stos+2>=sfree,
        sbc sfree+1             ;then stack-overflow
        bcs sterr               ;jump, if stack-overflow
;
;       check character$.
;
        lda #1                  ;get the address of character$
        jsr fndpar
        ldy #2                  ;current length must = 1
        lda (copy1),y           ;high byte must = 0
        bne argerr
        iny                     ;.y:=3
        lda (copy1),y           ;low byte must = 1
        cmp #1
        bne argerr
;
;       fetch character$(1:1)
;
        iny                     ;.y:=4
        lda (copy1),y           ;.a:=character$(1:1)
;
;       write character$(1:1) num times on the stack.
;
        ldy #0
        sty q1                  ;q1:=0 // loop variable
        sty q1+1
;
strlp   ldx num+1               ;while q1<>num do
        cpx q1+1
        bne str1
```

```
        ldx num
        cpx q1
        beq strok
;
str1    sta (stos),y          ;r$(q1:q1):=character$1:1)
;
        inc stos              ; tos:+1
        bne str2
        inc stos+1
;
str2    inc q1                ; q1:+1
        bne strlp
        inc q1+1
        jmp strlp             ;endwhile
;
;set the length of the string to num.
;
strok   lda num+1             ;save high byte of the length
        sta (stos),y
        iny                   ;.y:=1
        lda num               ;save low byte of the length
        sta (stos),y
;
        clc                   ;stos:+2 // room for the length //
        lda stos
        adc #<2
        sta stos
        lda stos+1
        adc #>2
        sta stos+1
        rts                   turn to COMAL with the result
;
argerr  ldx #1                ;"argument error"
        jmp runerr
;
sterr   ldx #56               ;"out of memory"
        jmp runerr
;
end     .end                  ;end of source text
```

Appendix A

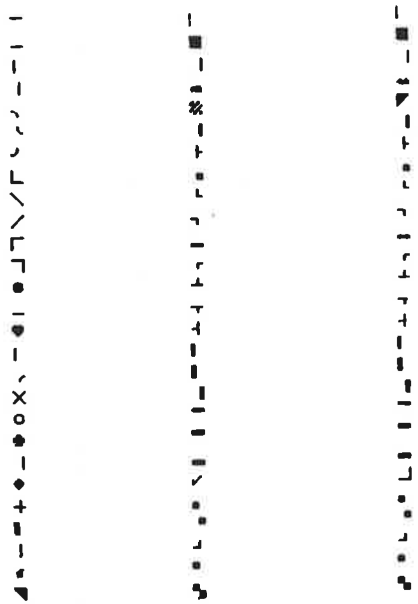
COMMODORE 64 Character Codes

	ASCII CHARACTERS mode		SCREEN CHARACTERS mode	
CODE	text	graphics	text	graphics
0			@	@
1			a	A
2			b	B
3	<STOP>		c	C
4			d	D
5	white		e	E
6			f	F
7			g	G
8	<SHIFT - C=> disable		h	H
9	<SHIFT - C=> enable		i	I
10	....		j	J
11	clear to end of line		k	K
12	form feed (printer)		l	L
13	<RETURN>		m	M
14	switch to lower case		n	N
15			o	O
16	....		p	P
17	cursor down		q	Q
18	reverse on		r	R
19	cursor home		s	S
20	<DEL>		t	T
21			u	U
22			v	V
23			w	W
24			x	X
25			y	Y
26			z	Z
27			æ	Æ
28	red		ø	Ø
29	cursor right		å	Å
30	green		↑	↑
31	blue		←	←
32	space			
33	!	!	!	!
34	"	"	"	"
35	#	#	#	#
36	$	$	$	$
37	%	%	%	%
38	&	&	&	&
39	'	'	'	'
40	(	(	(	(
41	)	)	)	)
42	*	*	*	*
43	+	+	+	+
44	,	,	,	,
45	-	-	-	-
46	.	.	.	.
47	/	/	/	/

	ASCII CHARACTERS mode		SCREEN CHARACTERS mode	
CODE	text	graphics	text	graphics
48	0	0	0	0
49	1	1	1	1
50	2	2	2	2
51	3	3	3	3
52	4	4	4	4
53	5	5	5	5
54	6	6	6	6
55	7	7	7	7
56	8	8	8	8
57	9	9	9	9
58	:	:	:	:
59	;	;	;	;
60	<	<	<	<
61	=	=	=	=
62	>	>	>	>
63	?	?	?	?
64	@	@	—	—
65	a	A	A	♠
66	b	B	B	│
67	c	C	C	—
68	d	D	D	—
69	e	E	E	—
70	f	F	F	—
71	g	G	G	│
72	h	H	H	│
73	i	I	I	╮
74	j	J	J	╰
75	k	K	K	╯
76	l	L	L	└
77	m	M	M	╲
78	n	N	N	╱
79	o	O	O	┌
80	p	P	P	┐
81	q	Q	Q	●
82	r	R	R	—
83	s	S	S	♥
84	t	T	T	│
85	u	U	U	╭
86	v	V	V	╳
87	w	W	W	○
88	x	X	X	♣
89	y	Y	Y	│
90	z	Z	Z	◆
91	æ	Æ	Æ	┼
92	ø	Ø	Ø	▌
93	å	Å	Å	│
94	↑	↑	▒	◤
95	←	←	▒	◥
96	—	—		
97	A	♠	▌	▌
98	B	│	▄	▄
99	C	—	▔	▔
100	D	—	▁	▁

ASCII CHARACTERS — mode / SCREEN CHARACTERS — mode

CODE	ASCII text	ASCII graphics	SCREEN text	SCREEN graphics
101	E			
102	F			
103	G			
104	H			
105	I			
106	J			
107	K			
108	L			
109	M			
110	N			
111	O			
112	P			
113	Q			
114	R			
115	S			
116	T			
117	U			
118	V			
119	W			
120	X			
121	Y			
122	Z			
123	[illegible]			
124	[illegible]			
125	A			
126	[illegible]			
127	[illegible]			
128				
129	orange			
130				
131	<RUN>			
132				
133	f1			
134	f3			
135	f5			
136	f7			
137	f2			
138	f4			
139	f6			
140	f8			
141	<SHIFT-RETURN>			
142	switch to upper case			
143				
144	black			
145	cursor up			
146	reverse off			
147	as <CLR> (clear screen)			
148	as <INST> (insert)			
149	brown			
150	light red			
151	dark grey			
152	grey			
153	light green			



Codes 128-255 are reversed images of codes 0-127

ASCII CHARACTERS CODE	mode text	graphics
154	light blue	
155	light grey	
156	purple	
157	cursor left	
158	yellow	
159	cyan	
160	space	
161		
162		
163		
164		
165		
166		
167		
168		
169		
170		
171		
172		
173		
174		
175		
176		
177		
178		
179		
180		
181		
182		
183		
184		
185		
186	✓	✓
187		
188		
189		
190		
191		
192	—	—
193	A	
194	B	
195	C	
196	D	
197	E	
198	F	
199	G	
200	H	
201	I	
202	J	
203	K	
204	L	
205	M	

SCREEN CHARACTERS CODE	mode text	graphics
206	N	
207	O	
208	P	
209	Q	
210	R	
211	S	
212	T	
213	U	
214	V	×
215	W	
216	X	
217	Y	
218	Z	
219		+
220		
221		
222		
223		
224		
225		
226		
227		
228		
229		
230		
231		
232		
233		
234		
235		
236		
237		
238		
239		
240		
241		
242		
243		
244		
245		
246		
247		
248		
249		
250	✓	
251		
252		
253		
254		
255		

Appendix B

Color Codes

Color code	Color	Grey scale	ASCII value	Keyboard
0	black	4/4	144	<CTRL-1>
1	white	0/4	5	<CTRL-2>
2	red	3/4	28	<CTRL-3>
3	cyan	1/4	159	<CTRL-4>
4	purple	2/4	156	<CTRL-5>
5	green	2/4	30	<CTRL-6>
6	blue	3/4	31	<CTRL-7>
7	yellow	1/4	158	<CTRL-8>
8	orange	2/4	129	<C= 1>
9	brown	3/4	149	<C= 2>
10	pink	2/4	150	<C= 3>
11	dark grey	3/4	151	<C= 4>
12	grey	2/4	152	<C= 5>
13	light green	1/4	153	<C= 6>
14	light blue	2/4	154	<C= 7>
15	light grey	1/4	155	<C= 8>

Color combinations on the TV/Monitor
(from the **Commodore 64-Programmer's Reference Guide**)

How do the colors go together?
+ = very well
o = well
= poorly

screen color	text color code															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		+		+	+	o		+	+		+	+	+	+	+	+
1	+		+		+	+	+		o	+	o	+	+		+	+
2		+			o			+	+		+					o
3	+					o	+					o			o	
4	+	o														o
5	+	o		o								o		+		o
6	o	+		+										o	+	+
7	+		+				o		o	+	o	+	+			
8	o	+	+					+		+						o
9		+						+	+		+					+
10	o	o	+					o		+						o
11	+	+		o				+					+	+	o	+
12	+	+	o				o			o		+				+
13	+					+	o					+				
14	+	+		+			+					o				o
15	+	+	+		o	o	+			o	o	+	+		o	

Appendix C

Calculations with COMAL

The COMAL operating system can handle 4 types of numerical constants and variables:

real numbers E.g. 3.232 , 4.6e-12 , PI, a , sum
integers 71 , -3067, nr# , item#
hexadecimal numbers $1a , $d7 , $ac00 , no , position
binary numbers %1011 , %10011010 , byte , id

Number ranges:
2.93873588e-39 <= real number <= 1.70141183e+38
-32768 <= integer <= 32767
0 = $00 <= hexadecimal <= $ffff = 65535
0 = %0 <752 binary number <= %1111111111111111 = 65535

Calculations are carried out according to the following rules:
An *expression* to be evaluated may contain a mix of all number types and number variables. It may contain a mix of arithmetic operators, relational operators and boolean operators. Standard COMAL functions and user defined functions can also be included:

* an expression is evaluated from left to right,
* however, various operators have different priority. The calculations are carried out according to the following priority, highest priority first:

PRIORITY:
(in order of highest priority)

1.	()	parentheses	
Arithmetic operators:			
2.	↑	exponentiation	2↑3 equals 8
3.	*	multiplication	2*3 equals 6
3.	/	division	7/2 equals 3.5
3.	DIV	integer division	54 DIV 8 equals 6
3.	MOD	remainder after division	23 MOD 7 equals 2
4.	+	addition	2+3 equals 5
4.	-	subtraction	4-3 equals 1
4.	-	monadic subtraction	-5+2 equals -3

Logical operators for bitwise comparisons:
(See further explanations in the reference section, Chapter 4.):

5.	BITAND	bitwise logical 'and'
5.	BITOR	bitwise logical 'or'
5.	BITXOR	bitwise logical 'exclusive or'

Relational operators:
(Comparisons occur in logical expressions, which can be TRUE (=1), if the comparison is true. Otherwise the logical expression has the value FALSE (=0)).

6.	<	less than	3*2<9 equals TRUE
6.	<=	less than or equal to	4*3<=10 equals FALSE
6.	=	equal to	1=2 equals FALSE
6.	>=	greater than or equal to	17>3 equals TRUE
6.	>	greater than	7>7 equals FALSE
6.	<>	not equal to	3*2<>6.01 equals TRUE

Boolean (logical) operators:
(See further explanation of the individual words in Chapter 4.):

7.	NOT	logical negation
8.	AND	logical 'and'
8.	AND THEN	as AND
9.	OR	logical 'or'
9.	OR ELSE	as OR

Standard functions:

INT(x)	Integer part of x	INT(3.2) equals 3
ABS(x)	Numerical value of x	ABS(-2.5) equals 2.5
SGN(x)	Sign of x	SGN(-3) equals -1
SIN(x)	Sine of x	SIN(PI/6) equals 0.5
COS(x)	Cosine of x	COS(PI) equals -1
TAN(x)	Tangent of x	TAN(PI/4) equals 1
ATN(x)	Inverse tangent of x	ATN(1) equals PI/4
LOG(x)	Natural logarithm	LOG(10) equals 2.3026
EXP(x)	Exponential function	EXP(2) equals 7.389
SQR(x)	Square root of x	SQR(9) equals 3

Examples of user defined functions:

```
FUNC asin(x)
  IF ABS(X)=1 THEN
   RETURN X*PI/2
  ELSE
   RETURN ATN(x/SQR(1-x*x))
   ENDIF
ENDIF

FUNC log10(x)
  RETURN LOG(x)/LOG(10)
ENDFUNC log10
```

Appendix D

Keyboard and the Screen Editor

The action of special keys in COMAL:

<←>
Underlining

<CTRL>
has special meaning when used with other keys. See the following.

<RUN/STOP>
interrupts program execution.
Action is affected by the COMAL statement ESC. See Chapter 4.

<SHIFT/LOCK>
locks <SHIFT> in upper case mode.
Release by pressing the key again.

<SHIFT>
As on a typewriter. If this key is held down while another key is pressed, an upper case character is produced. Letters appear as upper case. In the semigraphics mode the symbols on the right front side of the keys are produced. <SHIFT> pressed together with other special keys has other functions as described with these keys.

<C=> The Commodore Key:

<**C**= SHIFT>
Each activation toggles the screen display between lower and upper case.

<C= number>
Pressing the C= key with a number 1-8 switches to colors with color codes 8-15.

<C= graphics symbol>
Pressing a key with graphics symbols equals the symbol shown on the front left of the key.

<CLR/HOME>
moves the cursor to the upper left corner of the screen.

<SHIFT-CLR/HOME>
clears the screen.

<INST/DEL>
is the delete key. It deletes the character immediately to the left of the cursor, and the remainder of the line moves one space to fill in the gap.

<SHIFT-INST/DEL>
is the insert key. It pushes the character under the cursor and the rest of the line one space to the right.

<STOP-RESTORE>
If the <STOP> and <RESTORE> keys are pressed at the same time, the computer is 'reset'. The program in working memory is not lost.

<RETURN>
Indicates that all information on the current line should be interpreted and processed.

<CRSR>
There are two keys which are used to move the cursor around the screen. The arrows indicate directions. Each key has two functions. The function changes when the <SHIFT> key is depressed.

The Function Keys (<f1> - <f8>):
The *function keys* can be programmed by the user to perform various functions. (See further details in Chapter 5 in the section dealing with the procedure **defkey** in the COMAL package **system**.)

When COMAL is started up, these keys have the following functions:

<**f1**> RENUM + <RETURN
<**f2**> MOUNT + <RETURN>
<**f3**> USE turtle + <RETURN>
<**f4**> AUTO
<**f5**> EDIT
<**f6**> LIST
<**f7**> RUN + <RETURN> + CHR$(11) + <RETURN>

Note on <f7>: In addition to ordinary running of a program, this key can be used to start a program directly from the disk catalogue. RUN, RETURN would have the effect of running the program with the name which follows on the same line. However the text **prg** also appears after the program name when the catalogue is displayed, so the system reacts with an error message, placing the cursor just ahead of the 'error' **prg**.

Then ASCII-code 11 deletes the rest of the line. Now the line is correct, and the program can be run when the last RETURN is activated.

<f8> SCAN + <RETURN>
During program execution the function keys have other values: ASCII values 133 - 140.

After execution of one of the instructions **USE graphics** or **USE turtle** the function keys <f1>, <f3> and <f5> have the following meaning:

<f1>	**textscreen**	(show the text screen)
<f3>	**splitscreen**	(show graphics screen with 4 lines of text)
<f5>	**graphicscreen**	(show the graphics screen)

The Control key <CTRL>:

<CTRL-number>
<CTRL> together with a number 1 - 8 causes subsequent text to be written with the color indicated on the front of the number key. <CTRL> together with 9 or 0 toggles inverse text.

See also Appendix B on colors and Chapter 5 on the procedure **quote'mode** in the COMAL package **system**.

During editing of COMAL programs the following CTRL-functions are useful:

<CTRL> + **<letter>**
<CTRL-A>: Is used during the correction of a program line which extends over more than one line on the screen. If the first 1 to 4 characters in the linie in which the cursor is located is a line number, then the line number will be rewritten with no gaps. <CTRL-A> can also be used as an OOPS!-key: If a correction has been made, and <RETURN> has not yet been pressed, then pressing <CTRL-A> will cause the line to be printed again in its original form.

<CTRL-B>:	moves the cursor back one word.
<CTRL-C>:	corresponds to <STOP>.
<CTRL-D>:	dumps the graphics page to the printer. The printout begins 13 characters from the edge of the paper. This instruction can only be used with Commodore MPS801 compatible matrix printers.
<CTRL-E>:	changes the cursor color to white.
<CTRL-F>:	moves the cursor forward one word.
<CTRL-K>:	deletes all characters from the cursor position to the end of the line.

<CTRL-L>:	moves the cursor to just after the last non-blank character on the line.
<CTRL-M>:	corresponds to <RETURN>.
<CTRL-P>:	Executes a **hardcopy("lp:")**. I.e. prints out the text screen to the printer. The printout begins with a carriage return.
<CTRL-S>:	corresponds to <CLR/HOME>.
<CTRL-U>:	toggles the graphics mode functions for <f1>, <f3> og <f5>. See also the description of the function keys.
<CTRL-V>:	sets up the color choice **textcolors(6,6,1)**. This corresponds to a blue edge, blue background and subsequent white text. This is a good choice for a color display. Note that the current text screen is cleared by this instruction.
<CTRL-W>:	sets up the color choice as **textcolors(11,15,0)**. This corresponds to a dark grey border, light grey background and subsequent black text. This instruction clears the text screen. It is a good choice when using a black/white display.
<CTRL-X>:	changes the border color... It is followed by a color choice: <CTRL number> or <C= number>.
<CTRL-Y>:	changes the background color... It is followed by a color choice: <CTRL number> or <C= number>.
<CTRL-Z>:	The selected combination of border, screen and text colors are stored and will be reset when <STOP-RESTORE> is executed.

Appendix E

Handling Text with COMAL

Text variables (also called 'strings' or 'string variables') are specified in COMAL by means of a sequence of up to 80 characters followed by a $ sign. The first character must always be a letter, and certain special characters may not be included in the name.

Examples: name$, text$, from$, long'name$.

Before a text variable can be used, it must be declared (dimensioned). The system must be provided with information on the maximum number of characters the text variable will contain, so that room can be reserved in memory. This is done using the DIM statement:

Examples:

```
DIM text$ OF 80
DIM name$ OF 20
DIM answer$ OF 1
```

A text variable can contain any character sequence up to the dimensioned length. (Exception: the character " may not be used alone. If this character is to be included, you must use "" to indicate it. If a number is enclosed within the " ", then the corresponding ASCII code will be part of the text variable assignment.)

If a text variable is not dimensioned, then the first assignment instruction will automatically execute: **DIM name$ OF 40.** If a variable name is not dimensioned, and the name is used before an assignment has been made, then an error message will be generated.

Examples of text variable usage:

Make the assignments

slogan$:="comal is ok"
text$:="a flower is beautiful".

The text can be analyzed with the aid of standard functions and operators

length:=LEN(slogan$)	**length** is assigned the value 11, for **slogan$** consists of 11 characters. See a detailed description of the function LEN in Chapter 4.
position:="mal" IN slogan$	**position** is assigned the value 3, since the text **"mal"** is contained in **slogan$**, and the first character in **"mal"** is the 3. character in **slogan$**. See the more detailed description of the operator IN in Chapter 4.
ascii:=ORD(text$)	**ascii** is assigned the Commodore ASCII value for the letter **a** (= 65). See the ASCII values for all characters in Appendix A.
text$<slogan$	the logical expression will be true (TRUE = 1), because **a** precedes **c** in the alphabet.

Selection of String Segments:

letter$:=text$(8) or **letter$:=text$(8:8)**	**letter$** is assigned the string **"r"**, or which is the 8. character in **text$**
first$:=slogan$(:5)	**first$** is assigned the string **"comal"**, i.e. the 5 first characters in **slogan$**.
last$:=text$(13:) or **last$:=**text**$(LEN(text$)-8:)**	**last$** is assigned the text or **"beautiful"**, i.e. the last nine characters in **text$**.
t$:=slogan$(3:8)	**t$** is assigned the string **"mal is"**.
t$:="programs"(5:7)	**t$** is assigned the string **ram**.
t$:=STR$(1789)(2:3)	**t$** is assigned the string **"78"**.
t$:=(text$(4:9))(2:4)	**t$** is assigned the string **"owe"**, which is part of a part of a string.
text$(3:8):="bee"	**text$** will equal **a bee is beautiful** after this instruction has been executed.

Selection of text segments from indexed string variables:

DIM name$(3) OF 20
name$(1):="Adam Smith"
name$(2):="Eva Smith"
name$(3):="Krystle Smith"

t$:=name$(2)(1:5) — **t$** is assigned the string **"Eva S"**.

DIM item**$(3,2) OF 10**
item$(1,1):="book"
item$(1,2):="magazine"
item$(2,1):="car"
item$(2,2):="train"
item$(3,1):="oil"
item$(3,2):="gas"

select$:=item$(2,1)(2:3) — **select$** is assigned the string **"ar"**.

Concatenation of strings:

place$:="Yankee"=" stadium" — strings can be linked together using the character +.

message$:=slogan$+" and easy" — **message$** is assigned the string **"comal is ok and easy"**.

hello$:=name$(2)(:3)+text$(9:)+" and "+slogan$(10:11)

hello$ is assigned the string **"Eva is beautiful and ok"**.

t$:=("we and "=slogan$(1:5))(4:8) — **t$** is assigned the string **"and c"**.

String functions:

The user can define string functions at will to produce string segments:

```
0010 FUNC upper$(lower$)
0020   FOR l#:=1 TO LEN(lower$) DO
0030     a:=ORD(lower$(l#))
0040     IF a>64 AND a<94 THEN
0050       a:+128
0060       lower$(l#):=CHR$(a)
0070     ENDIF
0080   ENDFOR l#
0090   RETURN lower$
0100 ENDFUNC upper$
```

Examples of the use of the function **upper$**:
PRINT upper$("merry christmas") yields the printout:
MERRY CHRISTMAS

PRINT upper$("headline:")(4:8) gives the printout:
DLINE

Using COMAL it is easy to define the Basic-function **mid$**:

```
0010 FUNC mid$(a$,start,number)
0020   RETURN a$(start:start+number-1)
0030 ENDFUNC mid$
```

This function can be used in lieu of **mid$**, if you wish to use parts of existing Basic programs.

Appendix F

COMAL Error Numbers and Messages

The standard version of Commodore 64 COMAL contains error messages in two languages. When the computer is turned on with the COMAL cartridge in place, English error messages will be in effect. If desired Danish error messages can be selected by means of the order:

USE dansk

To get back to English, execute:

USE english

After issuing one of these orders, all subsequent error messages will be printed in the language you have chosen. However, error messages for the disk operating system will always be in English.

It is of course possible to incorporate error messages in other languages into a COMAL cartridge. Contact your Commodore national distribution center for further information.

The COMAL system can give error messages in the following situations:

- When typing in an instruction line
- When examining program structure (using **scan**)
- During a run (run-time errors)

The remainder of this Appendix includes a list of all error messages and their corresponding numerical code. Note that the list is given both in English and in Danish for those of you who may be curious about the strange language which COMAL can use:

Dynamic Syntax Error Messages:

<language element> not expected
too much on this line
<language element> ikke forventet

<language element> missing
more language elements are expected on this line
<language element> mangler

<language element 1> expected, not <language element 2>
<language element 1> forventet, ikke <language element 2>

Dynamic Structure Error Messages (Prepass):

<statement 1> without <statement 2>
<statement 1> uden <statement 2>

<statement> missing
<statement> mangler

<statement 1> expected, not <statement 2>
open- and close statements do not fit together
<statement 1> forventet, ikke <statement 2>

<statement> not allowed in control structures
DIM, DATA, IMPORT, PROC and FUNC are not allowed in control structures
<statement> ikke tilladt i styrestrukturer

import allowed in closed proc/func only
import kun tilladt i lukket proc/func

wrong type of <statement>
E.g. 1/ Text in WHEN-line expecting numeric expression.
2/ Variable names in FOR and ENDFOR differ
forkert slags <statement>

wrong name in <statement>
ENDFOR, ENDPROC and ENDFUNC must use same name as in FOR, PROC and FUNC respectively
forkert navn i <statement>

<name>: name already defined
The same name must not refer to different variable types inside the same scope. E.g. a, a#, a$
<name>: navn allerede defineret

<name>: unknown label
A label is missing within the current scope. E.g. A GOTO statement inside a procedure cannot refer to a label outside the procedure
<name>: ukendt etikette

illegal goto
You cannot jump into a structure by means of GOTO
ulovlig goto

Dynamic Run Time Error Messages:

<name>: unknown statement or procedure *E.g. Call of a package procedure without previous activation of the package by: USE package name*
<name>: ukendt statement eller procedure.

<name>: not a procedure
The name is a variable, package, function or a label, but not a procedure.
<name>: ikke en procedure

<name>: unknown variable
You have not assigned a value to the variable inside this scope.
<name>: ukendt variabel

<name>: wrong type
E.g. 1/ The variable is a string variable, but you try to use it as a numeric variable. 2/ Remember that RESTORE label: must be positioned immediately before a DATA line.
<name>: forkert type

<name>: wrong function type
<name>: forkert funktionstype

<name>: not an array nor a function
The name might be a simple variable, a procedure ..?
<name>: hverken tabel eller funktion

<name>: not a simple variable
The name might be an array, a procedure ..?
<name>: ikke en simpel variabel

<name>: unknown array or function
The name has not yet been dimensioned or defined.
<name>: ukendt tabel eller funktion

<name>: wrong array type
E.g. You try to refer to a two-dimensional array as though it is a one-dimensional array.
<name>: forkert tabeltype

<name>: import error
The name in the IMPORT statement is unknown or the type is wrong.
<name>: import fejl

<name>: unknown package
The COMAL system does not recognize the package name in USE. Remember that packages from diskette must be LINK'ed.
<name>: ukendt pakke

<name>: array redefined
<name>: navn redefineret

<name>: name already defined
<name>: navn allerede defineret

<name>: string not dimensioned
<name>: tekstvariabel ikke defineret

<name>: not a package
<name>: ikke en pakke

RUN TIME ERROR, WHICH CAN BE TRAP'PED:

0 report error
report fejl

1 argument error
E.g. 1/ Square root of a negative number. 2/ LOG to a non positive number
argument fejl

2 overflow
A number is too big. See Appendix C
overloeb

3 division by zero
division med nul

4 substring error
E.g. 1/ a$:=text$(from:to) requires 1 <=from<=to+1.
2/ text$(from:to):=a$ requires to<=LEN(text$)
deltekst-fejl

5 value out of range
uden for vaerdiomraade

6 step = 0
step = 0

7 illegal bound
DIM statements require: lower limit<=upper limit
ulovlige graenser

8 error in print using
The format is missing or has wrong syntax.
fejl i print using

10 index out of range
Index exceeds the limits from the DIM-statement.
ulovlig indexvaerdi

11 invalid file name
A file name must not exceed 69 characters
ulovligt filnavn

13 verify error
The program on disk and the program in memory differ. Remember that new names and typing errors change the program in memory.
verify fejl

14 program too big
program for stort

15 bad comal code
The program file has been changed, or transmission error
daarlig comalkode

16 not comal program file
Possibly the file is a Basic program file?
ej save-fil

17 program for other comal version
program til anden comalversion

18 unknown file attribute
ukendt filattribut

30 invalid color
-1<=color code<=15
ulovlig farve

31 invalid boundary
In viewport: 0<=vxmin<=vxmax<=319;
0<=vymin<=vymax<=199
ulovlig graense

32 invalid shape number
0<=shape number<=47 (or 46 if turtle-sprite is visible)
ulovlig tegning-nummer

33 shape length must be 64
tegningens laengde skal vaere 64

34 invalid sprite number
*0<=*sprite number<=7 (or 6 if turtle is visible)
ulovlig sprite-nummer

35 invalid voice
1<=voice<=3
ulovlig stemme

36 invalid note
See Index for note and frequency.
ulovlig node

Run Time Errors, Which cannot be TRAP'ped:

51 system error
Serious error in COMAL system. Try the NEW command.
system fejl

52 out of memory
Memory shortage for program, names, data and function calls. All memory is released except for the program memory. E.g. too many recursive calls.
for lidt hukommelse

53 wrong dimension in parameter
The actual and the formal parameters in a procedure call must have the same dimension.
forkert dimension i parameter

54 parameter must be an array
If the formal parmeter is an array, must be also the actual parameter
parameter skal vaere en tabel

55 too few indices
The array is called with too few indices.
for faa indices

56 cannot assign variable
E.g. You have tried to assign a string to a name which is not a string variable.
kan ikke tildele variabel

57 ikke implementeret
not implemented

58 con not possible
CON is not allowed when:
1) the computer is just switched on
2) after NEW, LINK, DISCARD or SCAN
3) after an error
4) interruption of a command (ex. a procedure call)
5) the program has terminated with END
6) the program has been revised
7) a new name is added
con ikke mulig

59 program has been modified
E.g. After a procedure modification, a RUN or a SCAN must precede a call as a direct command to the procedure.
programmet er blevet modificeret

60 too many indices
for mange indices

61 function value not returned
A RETURN statement has not been executed.
funktionsvaerdi ikke returneret

62 not a variable
ikke en variabel

67 parameter lists differ or not closed
The external procedure must be CLOSED, and the parameters must match the ones of the call.
parameterlister afviger eller ikke lukket

68 no close wrong parameter type
The parameter types of the procedure call do not match the parameters in the procedure heading.
forkert parametertype

73 non-ram load
An attempt has been made to store a package in occupied RAM memory.
ikke-ram indlaesning

74 checksum error in object file
An error in the LINK'ed file
checksumfejl i objektfil

75 memory area is protected
An attempt has been made to LINK a modul into the area of another package or the COMAL program.
hukommelsesomraade beskyttet

76 too many libraries
A package modul is often called a library; the number of libraries<=10, but there are no limits on the number of packages.
for mange biblioteker

77 not an object file
The attemped LINK'ed file is not an object code file
ikke en objektfil

78 no matching when
A CASE-expression matches no WHEN line. Add an OTHERWISE line.
ingen passende when

79 too many parameters
The procedure call contains too may parameters.
for mange parametre

Syntax Errors:

101 syntax error
The COMAL system cannot find a more appropriate error message.
syntaksfejl

102 wrong type
The statement contains an expression of the wrong type.
forkert type

103 statement too long or too complicated
saetning for lang eller for kompliceret

104 statement only, not command
kun som saetning, ikke som kommando

106 line number range: 1 to 9999
linienumre er fra 1 til 9999

108 procedure/function does not exist
procedure/funktion findes ikke

109 structured statement not allowed here
A structured statement is not allowed in single line versions of IF- , FOR-, WHILE- or REPEAT statements.
struktureret saetning ikke tilladt her

110 not a statement
The first character is not a valid character for a statement on this line.
ikke en saetning

111 line numbers will exceed 9999
If AUTO, RENUM or MERGE continues, line numbers will exceed 9999.
linienumre vil overskride 9999

112 source protected!!!
In COMAL, lines can be protected against LIST'ing. See program on Demo-disk.
kilde beskyttet!!!

113 illegal character
The symbol cannot begin with this character.
ulovligt tegn

114 error in constant
The syntax for the real-, binary or hexadecimal contstant is wrong.
fejl i konstant

115 error in exponent
The syntax of the exponent is wrong.
fejl i eksponent

Input/Output- Error Messages, Which can All be TRAP'ped:

200 end of data
An attempt has been made to read past the last DATA value.
ikke flere datalinier

201 end of file
An attempt has been made to read past the last record in a sequential file.
slut paa fil

202 file already open
A file with the same stream number has already been opened.
fil allerede aaben

203 file not open
fil ikke aaben

204 not input file
You cannot read a file, which has been opened with WRITE.
ikke en inputfil

205 not output file
You cannot write to a file, which has been opened with READ.
ikke en outputfil

206 numeric constant expected
An attempt to read a non-numeric value has been made.
numerisk konstant forventet

207 not random access file
As the file has been opened as a sequential file, you cannot address an individual record.
ikke en fil med direkte tilgang

208 device not present
The chosen device has not yet been connected to the serial bus.
enhed ikke tilstede

209 too many files open
No more than 9 files may be opened at the same time. Only 1 random access file may be opened at a time
for mange filer aabne

210 read error
During read-in from the serial bus there has been no answer before time-out.
laesefejl

211 write error
During print-out to the serial bus there has been no answer before time-out.
skrivefejl

212 short block on tape
(kort blok paa band)

213 long block on tape
lang blok paa baand

214 checksum error on tape
checksumfejl paa baand

215 end of tape
slut paa baand

216 file not found
fil ikke fundet

217 unknown device
ukendt enhed

218 illegal operation
ulovlig operation

219 i/o break
i/o afbrydelse

MESSAGES FROM THE DISK OPERATING SYSTEM (ONLY IN ENGLISH):

222 read error (The data block is not present.)

223 read error (Checksum error in the data block)

224	read error (Error in byte decoding)
225	write error (Write/read error)
226	write protect on (The diskette is write protected.)
227	read error (Checksum error in the header)
228	write error (Long data block)
229	disk id mismatch (UnMOUNTED or nonmatching diskette)
230	syntax error (Ordinary syntax error)
231	syntax error (Incorrect DOS-command)
232	syntax error (Line too long)
233	syntax error (Incorrect file name)
234	syntax error (No file was indicated)
239	syntax error (Incorrect pass-command)
250	record not present (Reading beyond the last record)
251	overflow in record (Record length overrun)
252	file too large (No room for the random file)
260	write file open (An already opened file opened again)
261	file not open (Tried to access an unopened file)
262	file not found (The file does not exist in the disk drive.)
263	file exists (The file is already present on the disk.)
264	file type mismatch (Operation on files of different type)
265	no block (The block is reserved.)
266	illegal track and sector (Track/sector does not exist.)
267	illegal system t or s (Illegal system track or sector)

270	no channel (There is no available channel.)
271	dir error (Directory error)
272	disk full (The diskette is filled up.)
273	cbm dos vx.x yyyy (Diskette status)
274	drive not ready (No diskette)

Appendix G

User Comments and Corrections

These pages are intended to be used for your comments and corrections. The authors and publishers of this manual will be pleased to learn about your comments. It will be advantageous to all users that errors are documented and corrected.

Thanks for your help!

Appendix H

Sample COMAL Programs

Music Programs

```
0010 // save "@Music 1"
0020 DIM code$ OF 3
0030 USE sound
0040
0050 LOOP
0060   PAGE
0070   PRINT "Choose voice (1,2 or 3)"
0080   PRINT "Choose note  (a2,c4,b3,...)"
0090   PRINT "The numbers = octave:"
0100   PRINT "'c4' is middle C (4. octave - 440 Hz)"
0110   PRINT "'f5#' is 'f sharp' in the octave above"
0120   PRINT AT 22,1: "LESSON 1: We play a single note..."
0130   PRINT AT 20,1: "(Press <RUN/STOP> to end ...)"
0140   PRINT
0150   INPUT AT 8,1: "voice: ": voice
0160   INPUT AT 9,1: "note-code: ": code$
0170   play(1,code$)
0180 ENDLOOP
0190
0200 PROC play(voice,code$)
0210   IF code$<>"z" THEN
0220     note(voice,code$)
0230     gate(voice,1) // attack and decay
0240   ENDIF
0250   pause(16) // sustain
0260   gate(voice,0) // release
0270 ENDPROC play
0280
0290 PROC pause(sec'32)
0300   TIME 0
0310   WHILE TIME<1.875*sec'32 DO NULL
0320 ENDPROC pause
```

```
0010 // save "@Music 2"
0020 DIM code$ OF 3
0030 USE sound
0040
0050 LOOP
0060   PAGE
0070   PRINT "Type in a note (a2,b5,c4,...)"
0080   PRINT "The 3 voices are played in succession."
0090   PRINT AT 22,1: "LESSON 2: 3 voices are played..."
0100   PRINT AT 20,1: "Press <RUN/STOP> to end..."
0110   PRINT
0120
0130   FOR voice:=1 TO 3 DO
0140     soundtype(voice,3)
0150   ENDFOR voice
```

```
0160
0170   INPUT AT 7,1: "note-code: ": code$
0180
0190   FOR voice:=1 TO 3 DO
0200     PRINT AT 10,1: "voice ";voice
0210     play(voice,code$)
0220     play(voice,"z")
0230   ENDFOR voice
0240
0250 ENDLOOP
0260
0270 PROC play(voice,code$)
0280   IF code$<>"z" THEN
0290     note(voice,code$)
0300     gate(voice,1) // attach and decay
0310   ENDIF
0320   pause(8) // sustain
0330   gate(voice,0) // release
0340 ENDPROC play
0350
0360 PROC pause(sec'32)
0370   TIME 0
0380   WHILE TIME<1.875*sec'32 DO NULL
0390 ENDPROC pause
```

```
0010 // save "@Music 3"
0020 DIM code$ OF 2, answer$ OF 5
0030 USE sound
0040
0050 LOOP
0060   PAGE
0070   PRINT "Let's play some notes together"
0080   PRINT "and create a simple melody..."
0090   PRINT AT 22,1: "LESSON 3: We play a melody..."
0100
0110   FOR voice:=1 TO 3 DO
0120     soundtype(voice,3)
0130   ENDFOR voice
0140
0150   INPUT AT 4,1: "continue or end (c/e)? ": answer$
0160   IF answer$="e" THEN STOP
0170   INPUT AT 6,1: "voice (1/2/3)? ": voice
0180
0190   play'melody
0200
0210 ENDLOOP
0220
0230 PROC play(voice,code$)
0240   IF code$<>"z" THEN
0250     note(voice,code$)
0260     gate(voice,1) // attack and decay
0270   ENDIF
0280   pause(tid) // sustain
0290   gate(voice,0) // release
0300 ENDPROC play
0310
0320
0330 PROC play'melody // Row, Row, Row Your Boat
0340
0350 melody:
0360   DATA "c4",8,"z",2,"c4",8,"z",2,"c4",8,"d4",4
0370   DATA "e4",8,"z",8,"e4",8,"d4",4,"e4",8
0380   DATA "f4",4,"g4",16,"z",8,"c5",4
```

```
0390   DATA "c5",4,"c5",4,"g4",4,"g4",4
0400   DATA "g4",4,"e4",4,"e4",4,"e4",4
0410   DATA "c4",4,"c4",4,"c4",4,"z",8,"g4",8
0420   DATA "f4",4,"e4",8,"d4",4,"c4",8
0430
0440   RESTORE melody
0450   WHILE NOT EOD DO
0460     READ code$,tid
0470     play(voice,code$)
0480   ENDWHILE
0490
0500 ENDPROC play'melody
0510
0520 PROC pause(sec'32)
0530   TIME 0
0540   WHILE TIME<1.875*sec'32 DO NULL
0550 ENDPROC pause
```

```
0010 // save "@Music 4"
0020 DIM code$ OF 2
0030 USE sound
0040
0050 LOOP
0060
0070   PAGE
0080   PRINT AT 22,1: "LESSON 4: Sound level, type and ADSR..."
0090   PRINT AT 1,1: "Sound level and sound type can be"
0100   PRINT "selected for each voice."
0110   PRINT
0120   PRINT "Choose the parameters in SOUNDTYPE,"
0130   PRINT "and choose the ADSR values..."
0140   PRINT
0150   PRINT "Your choices will remain valid until"
0160   PRINT "the parameters are redefined."
0170
0180   INPUT AT 11,1: "VOICE     (1/2/3)? ": voice
0190   INPUT AT 13,1: "VOLUME    (0-15)?   ": vol
0200   INPUT AT 15,1: "SOUNDTYPE (1/2/3/4)? ": type
0210   soundtype(voice,type)
0220   volume(vol)
0230   PAGE
0240   PRINT "Voice:";voice;" - Sound type:";type
0250   PRINT "The sound level is";vol;"."
0260   PRINT
0270   PRINT "----------------------------------------"
0280   PRINT "ADSR parameters: attack, decay,"
0290   PRINT "sustain and release are chosen..."
0300   PRINT
0310   PRINT
0320   PRINT "    *"
0330   PRINT "   * *            Each parameter can"
0340   PRINT "  *   *****       vary from 0 to 15."
0350   PRINT " *         *"
0360   PRINT "*           *"
0370   PRINT "  A  D  S  R "
0380   PRINT
0390   PRINT "A: attack time       D: decay time"
0400   PRINT "S: sustain level     R: release time"
0410   PRINT "----------------------------------------"
0420   INPUT AT 21,1: "A,D,S,R? ": a,d,s,r
0430   adsr(voice,a,d,s,r)
0440
0450   play'melody
0460
```

```
0470 ENDLOOP
0480
0490 PROC play(voice,code$)
0500   IF code$<>"z" THEN
0510     note(voice,code$)
0520     gate(voice,1) // attack and decay
0530   ENDIF
0540   pause(tid) // sustain
0550   gate(voice,0) // release
0560 ENDPROC play
0570
0580 PROC play'melody // Row, Row, Row Your Boat
0590 melody:
0600   DATA "c4",8,"z",2,"c4",8,"z",2,"c4",8,"d4",4
0610   DATA "e4",8,"z",8,"e4",8,"d4",4,"e4",8
0620   DATA "f4",4,"g4",16,"z",8,"c5",4
0630   DATA "c5",4,"c5",4,"g4",4,"g4",4
0640   DATA "g4",4,"e4",4,"e4",4,"e4",4
0650   DATA "c4",4,"c4",4,"c4",4,"z",8,"g4",8
0660   DATA "f4",4,"e4",8,"d4",4,"c4",8
0670
0680   RESTORE melody
0690   WHILE NOT EOD DO
0700     READ code$,tid
0710     play(voice,code$)
0720   ENDWHILE
0730
0740 ENDPROC play'melody
0750
0760 PROC pause(sec'32)
0770   TIME 0
0780   WHILE TIME<1.875*sec'32 DO NULL
0790 ENDPROC pause
```

```
0010 // save "@Music 5"
0020 DIM code$ OF 3
0030 DIM tone#(50), ads'pause#(50), r'pause#(50)
0040 USE sound
0050 volume(15)
0060 soundtype(1,2)
0070 adsr(1,6,6,8,6)
0080
0090 no:=0
0100 WHILE NOT EOD DO
0110   no:+1
0120   READ code$,tim
0130   tone#(no):=frequency(code$)
0140   ads'pause#(no):=tim*2
0150   r'pause#(no):=tim*2
0160 ENDWHILE
0170
0180 tone#(no+1):=0
0190 setscore(1,tone#(),ads'pause#(),r'pause#())
0200 playscore(1,0,0)
0210
0220 number:=0
0230 WHILE NOT waitscore(1,0,0) DO
0240   number:+1
0250   PRINT number;
0260 ENDWHILE
0270 END
0280
0290 PROC pause(sec'32)
0300   TIME 0
```

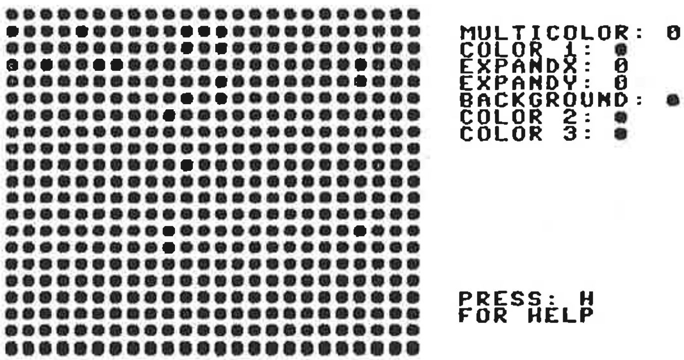
```
0310   WHILE TIME<1.875*sec'32 DO NULL
0320 ENDPROC pause
0330
0340 DATA "c4",8,"c4",8,"c4",8,"d4",4
0350 DATA "e4",8,"e4",8,"d4",4,"e4",8
0360 DATA "f4",4,"g4",16,"c5",4
0370 DATA "c5",4,"c5",4,"g4",4,"g4",4
0380 DATA "g4",4,"e4",4,"e4",4,"e4",4
0390 DATA "c4",4,"c4",4,"c4",4,"g4",8
0400 DATA "f4",4,"e4",8,"d4",4,"c4",8
```

Sprite Editor

The program SPRITEEDITOR is on the COMAL demonstration diskette (and tape). This program can be used to create sprite images. A drawing which has been prepared and saved using this program can later be loaded into another program using the order:

loadshape (<drawingno>,<filename$>)

The sprite editor program starts by displaying the following:



Each of the dots corresponds to a dot on the screen.

Movement of the drawing cursor from dot to dot is achieved using the cursor keys. The dots can be marked to indicate that they are to have a color different from the background color.

Choices are available from a menu shown on the right-hand side of the screen. If HELP is required, press H. A screen with user information will then appear.

Adress List

```
0010 // save "@Addr List Demo"
0020 DIM reply$ OF 1, name$(100) OF 40
0030 DIM street$(100) OF 40, city$(100) OF 40
0040 DIM phone$(100) OF 20, flag$ OF 40
0050 DIM searchkey$ OF 40, string$ OF 150
0060 number:=0 // number of records
0070 PAGE
0080 PRINT "This program illustrates the use of"
0090 PRINT "SEQUENTIAL FILES. It can be used to"
0100 PRINT "create a list of names, addresses"
0110 PRINT "and telephone numbers."
0120 PRINT "Each record will have the format:"
0130 PRINT
0140 PRINT "     name"
0150 PRINT "     street"
0160 PRINT "     city"
0170 PRINT "     phonenumber"
0180 PRINT
0190 PRINT
0200 PRINT "Press any key to continue..."
0210
0220 wait'for'keystroke
0230
0240 LOOP
0250   show'menu
0260   flag$:=""
0270   wait'for'keystroke
0280   CASE reply$ OF
0290   WHEN "1"
0300     load'file
0310   WHEN "2"
0320     create'record
0330   WHEN "3"
0340     list'file
0350   WHEN "4"
0360     search'file
0370   WHEN "5"
0380     sort'file
0390   WHEN "6"
0400     change'record
0410   WHEN "7"
0420     delete'record
0430   WHEN "8"
0440     save'file
0450   OTHERWISE
0460     PRINT "Illegal reply.."
0470     wait'for'keystroke
0480   ENDCASE
0490 ENDLOOP
0500
```

```
0510 PROC show'menu
0520   PAGE
0530   PRINT "-----===== MAIN MENU =====-----"
0540   PRINT
0550   PRINT
0560   PRINT "     (1) LOAD   the file"
0570   PRINT "     (2) CREATE a record"
0580   PRINT "     (3) LIST   the file"
0590   PRINT "     (4) SEARCH the file"
0600   PRINT "     (5) SORT   alphabetically"
0610   PRINT "     (6) CHANGE a record"
0620   PRINT "     (7) DELETE a record"
0630   PRINT "     (8) SAVE   revised file"
0640   PRINT
0650   PRINT
0660   PRINT "Records: ";number
0670   IF number=0 THEN flag$:="Please load or create a file..."
0680   PRINT
0690   PRINT flag$
0700 ENDPROC show'menu
0710
0720 PROC load'file
0730   OPEN FILE 1,"Addresses",READ
0740   INPUT FILE 1: number
0750   FOR no:=1 TO number DO
0760     INPUT FILE 1: name$(no)
0770     INPUT FILE 1: street$(no)
0780     INPUT FILE 1: city$(no)
0790     INPUT FILE 1: phone$(no)
0800   ENDFOR no
0810   CLOSE FILE 1
0820 ENDPROC load'file
0830
0840 PROC create'record
0850   PAGE
0860   PRINT "::::: CREATE A NEW RECORD :::::"
0870   PRINT
0880   PRINT
0890   IF number=100 THEN flag$:="No more room for data!"
0900   IF flag$="" THEN
0910     number:+1
0920     INPUT "Name    ": name$(number)
0930     INPUT "Street  ": street$(number)
0940     INPUT "City    ": city$(number)
0950     INPUT "Phone   ": phone$(number)
0960   ENDIF
0970 ENDPROC create'record
0980
0990 PROC list'file
1000   PAGE
1010   PRINT "::::: LISTING THE FILE :::::"
1020   PRINT
1030   IF number=0 THEN
1040     flag$:="No files in memory!"
1050     PRINT
1060   ELSE
1070     FOR no:=1 TO number DO print'record(no)
1080   ENDIF
1090 ENDPROC list'file
1100
1110 PROC search'file
1120   PAGE
1130   PRINT "::::: FILE SEARCH :::::"
1140   PRINT
1150   PRINT
1160   flag$:="I am searching..."
```

```
1170    INPUT "Search key: ": searchkey$
1180    FOR no:=1 TO number DO
1190      string$:=name$(no)+street$(no)+city$(no)+phone$(no)
1200      IF searchkey$ IN string$ THEN print'record(no)
1210    ENDFOR no
1220    flag$:=""
1230 ENDPROC search'file
1240
1250 PROC print'record(no)
1260    PRINT
1270    PRINT AT 0,10: "---------------(",no,")"
1280    PRINT AT 0,10: name$(no)
1290    PRINT AT 0,10: street$(no)
1300    PRINT AT 0,10: city$(no)
1310    PRINT AT 0,10: phone$(no)
1320    PRINT
1330    wait'for'keystroke
1340 ENDPROC print'record
1350
1360 PROC sort'file
1370    PAGE
1380    PRINT "::::: SORT BY NAME ALPHABETICALLY :::::"
1390    PRINT
1400    PRINT
1410
1420    PROC swap(REF a$,REF b$) CLOSED
1430      c$:=a$; a$:=b$; b$:=c$
1440    ENDPROC swap
1450
1460    REPEAT
1470      no'swap:=TRUE
1480      FOR no:=1 TO number-1 DO
1490        PRINT AT 10,1: "Sorting... ",no
1500        IF name$(no+1)<name$(no) THEN
1510          swap(name$(no),name$(no+1))
1520          swap(street$(no),street$(no+1))
1530          swap(city$(no),city$(no+1))
1540          swap(phone$(no),phone$(no+1))
1550          no'swap:=FALSE
1560        ENDIF
1570      ENDFOR no
1580    UNTIL no'swap
1590 ENDPROC sort'file
1600
1610 PROC change'record
1620    PAGE
1630    PRINT "::::: CHANGE A RECORD :::::"
1640    PRINT
1650    PRINT
1660    INPUT "Which record number? ": no
1670    IF no<=number THEN
1680      print'record(no)
1690      INPUT AT 14,1: "Is this the right record ? (y/n)? ": reply$
1700      PRINT
1710      PRINT
1720      IF reply$ IN "yY" THEN
1730        INPUT "Name   : ": name$(no)
1740        INPUT "Street : ": street$(no)
1750        INPUT "City   : ": city$(no)
1760        INPUT "Phone  : ": phone$(no)
1770      ENDIF
1780    ELSE
1790      flag$:="There are only "+STR$(number)+" records"
1800    ENDIF
1810 ENDPROC change'record
1820
```

```
1830 PROC delete'record
1840   PAGE
1850   PRINT ":::::: DELETE A RECORD :::::"
1860   PRINT
1870   PRINT
1880   INPUT "Which record number? ": record
1890   IF record>number THEN
1900     flag$:="Use a smaller record number!"
1910   ELSE
1920     print'record(record)
1930     PRINT
1940     INPUT "Is this the right record (y/n)? ": reply$
1950     PRINT
1960     IF reply$ IN "yY" THEN
1970       FOR no:=record TO number-1 DO
1980         name$(no):=name$(no+1)
1990         street$(no):=street$(no+1)
2000         city$(no):=city$(no+1)
2010         phone$(no):=phone$(no+1)
2020       ENDFOR no
2030       number:-1
2040     ENDIF
2050   ENDIF
2060 ENDPROC delete'record
2070
2080 PROC save'file
2090   PAGE
2100   PRINT ":::::: SAVING FILE TO DISK :::::"
2110   OPEN FILE 1,"@Addresses",WRITE
2120   PRINT FILE 1: STR$(number)
2130   PRINT
2140   PRINT
2150   FOR no:=1 TO number DO
2160     PRINT FILE 1: name$(no)
2170     PRINT FILE 1: street$(no)
2180     PRINT FILE 1: city$(no)
2190     PRINT FILE 1: phone$(no)
2200   ENDFOR no
2210   CLOSE FILE 1
2220 ENDPROC save'file
2230
2240 PROC wait'for'keystroke
2250   PRINT
2260   PRINT "< >...";
2270   REPEAT
2280     reply$:=KEY$
2290   UNTIL reply$<>CHR$(0)
2300   PRINT AT 0,2: reply$
2310 ENDPROC wait'for'keystroke
```

```
0010 // save "plotter demo"
0020
0030 DIM sc$ OF 1
0040
0050 setup'plotter
0060 //
0070 // MAIN PROGRAM
0080 //
0090 demo'size
0100 demo'color
0110 demo'case
0120 demo'rotation
0130
0140 square(100)
```

```
0150 blankline(2)
0160 dotlines(15)
0170 blank'line(8)
0180 circle(240,240,200)
0190 blank'line(14)
0200 spinsquares(150)
0210
0220 setup'plotter
0230
0240 END // MAIN PROGRAM
0250
0260
0270
0280 PROC demo'size
0290   FOR i:=0 TO 3 DO
0300     select'size(i)
0310     print'hello
0320     blank'line(1)
0330   ENDFOR i
0340 ENDPROC demo'size
0350
0360 PROC demo'color
0370   select'size(2)
0380   FOR i:=0 TO 3 DO
0390     switch'color(i)
0400     print'hello
0410   ENDFOR i
0420 ENDPROC demo'color
0430
0440 PROC demo'case
0450   blank'line(1)
0460   select'case(0) // upper case
0470   print'hello
0480   select'case(1) // lower case
0490   print'hello
0500 ENDPROC demo'case
0510
0520 PROC demo'rotation
0530   blank'line(2)
0540   rot'char(1)
0550   print'hello
0560   rot'char(0)
0570   print'hello
0580 ENDPROC demo'rotation
0590
0600 PROC dotlines(n)
0610   zero'pen("h")
0620   FOR i:=0 TO n DO
0630     plot("m",0,-i*20)
0640     dot'line(i)
0650     plot("d",400,-i*20)
0660   ENDFOR i
0670   blank'line(4)
0680   dot'line(0)
0690 ENDPROC dotlines
0700
0710 PROC circle(x0,y0,radius)
0720   plot("m",x0,y0)
0730   zero'pen("i")
0740   plot("r",radius,0)
0750   FOR v:=0 TO 360 STEP 5 DO
0760     t:=PI*v/180
0770     x:=radius*COS(t)
0780     y:=radius*SIN(t)
0790     plot("j",x,y)
0800   ENDFOR v
```

```
0810    blank'line(4)
0820 ENDPROC circle
0830
0840 PROC square(side)
0850    blank'line(3)
0860    plot("j",0,side)
0870    plot("j",side,side)
0880    plot("j",side,0)
0890    plot("j",0,0)
0900 ENDPROC square
0910
0920 PROC spinsquares(s)
0930    plot("m",240,240)
0940    zero'pen("i")
0950    FOR v:=0 TO 360 STEP 20 DO
0960      t:=PI*v/180
0970      draw'box(s,t)
0980    ENDFOR v
0990    blank'line(4)
1000 ENDPROC spinsquares
1010
1020 PROC draw'box(s,t)
1030    plot("j",s*COS(t),s*SIN(t))
1040    plot("j",s*SQR(2)*COS(t+PI/4),s*SQR(2)*SIN(t+PI/4))
1050    plot("j",s*COS(t+PI/2),s*SIN(t+PI/2))
1060    plot("j",0,0)
1070 ENDPROC draw'box
1080
1090 PROC blank'line(bl)
1100    plotter'on
1110    FOR i:=1 TO bl DO
1120      PRINT FILE 6:
1130    ENDFOR i
1140    plotter'off
1150 ENDPROC blank'line
1160
1170 PROC print'hello
1180    plotter'on
1190    PRINT FILE 6: "HELLO!"
1200    plotter'off
1210 ENDPROC print'hello
1220
1230
1240 // PLOTTER PROCEDURES
1250
1260 PROC plotter'on
1270    OPEN FILE 6,"u6:",WRITE
1280 ENDPROC plotter'on
1290
1300 PROC plotter'off
1310    CLOSE FILE 6
1320 ENDPROC plotter'off
1330
1340 PROC switch'color(pen)
1350    talk("2",STR$(pen))
1360 ENDPROC switch'color
1370
1380 PROC select'size(size)
1390    talk("3",STR$(size))
1400 ENDPROC select'size
1410
1420 PROC select'ascii
1430    talk("0","")
1440 ENDPROC select'ascii
1450
1460 PROC plot(sc$,x,y)
```

```
1470   talk("1",sc$+" "+STR$(x)+" "+STR$(y))
1480 ENDPROC plot
1490
1500 PROC zero'pen(zp$)
1510   // zp$ = h/i for abs/relative
1520   talk("1",zp$)
1530 ENDPROC zero'pen
1540
1550 PROC rot'char(rot)
1560   // rot=0/1 for hor/rot 90 deg CW
1570   talk("4",STR$(rot))
1580 ENDPROC rot'char
1590
1600 PROC dot'line(dash)
1610   // dash=0 to 15, 0 = unbroken
1620   talk("5",STR$(dash))
1630 ENDPROC dot'line
1640
1650 PROC select'case(nr)
1660   // nr=0/1 for upper/lower case
1670   talk("6",STR$(nr))
1680 ENDPROC select'case
1690
1700 PROC reset'plotter
1710   talk(7,"")
1720 ENDPROC reset'plotter
1730
1740 PROC setup'plotter
1750   select'case(1) // lower case
1760   switch'color(1) // blue
1770   rot'char(0) // horizontal
1780   dot'line(0) // unbroken
1790   select'size(1) // normal
1800 ENDPROC setup'plotter
1810
1820 PROC talk(sa$,text$)
1830   OPEN FILE 100,"u6:/s"+sa$,WRITE
1840   PRINT FILE 100: text$
1850   CLOSE FILE 100
1860 ENDPROC talk
```

```
0010 // save "@Train Demo"
0020
0030 PAGE
0040 PRINT AT 2,2: "ELECTRIC TRAIN DEMO"
0050 PRINT AT 4,2: "Your train should start at the station"
0060 PRINT AT 5,2: "with the passage detector just behind"
0070 PRINT AT 6,2: "the last car.  Start the train and then"
0080 PRINT AT 7,2: "press any key to turn control over"
0090 PRINT AT 8,2: "to your computer..."
0100 WHILE KEY$=CHR$(0) DO NULL
0110 PAGE
0120 PRINT AT 2,2: "ELECTRIC TRAIN DEMO"
0130
0140 // Port B bit 0 can be connected to the collector of a Darlington
0150 // Phototransistor.  The emitter is connected to ground.
0160 // Bit 0 will be low when the fototransistor is illuminated.
0170 // Port B bit 1 should be connected to a transistor and relay
0180 // so that bit 1 high starts the train.
0190
0200 // MAIN PROGRAM
0210
0220 define'variables
0230 set'port'b
```

```
0240 start'train
0250 print'list
0260
0270 REPEAT
0280   check'light
0290   delay(1.5)
0300   stop'train
0310   delay(10)
0320   start'train
0330 UNTIL KEY$<>""
0340 stop'train
0350 PAGE
0360 END "Au revoir!"
0370
0380 // ALL PROCEDURES FOLLOW BELOW
0390
0400 PROC print'list
0410   PRINT AT 12,4: "train running"
0420   PRINT AT 13,4: "train passes light"
0430   PRINT AT 14,4: "train waiting at station"
0440   PRINT AT 18,4: "Pressing any key will stop the train"
0450   PRINT AT 19,4: "next time it stops at the station..."
0460 ENDPROC print'list
0470
0480 PROC start'train
0490   POKE port'b,PEEK(port'b) BITOR 2
0500   advance'pointer
0510 ENDPROC start'train
0520
0530 PROC check'light
0540   WHILE PEEK(port'b) BITAND 1<>1 DO NULL
0550   advance'pointer
0560 ENDPROC check'light
0570
0580 PROC delay(sec)
0590   TIME 0
0600   WHILE TIME<sec*60 DO NULL
0610 ENDPROC delay
0620
0630 PROC stop'train
0640   POKE port'b,PEEK(port'b) BITAND 253
0650   advance'pointer
0660 ENDPROC stop'train
0670
0680 PROC define'variables
0690   port'b:=$dd01
0700   port'b'ddr:=$dd03
0710   position:=1
0720 ENDPROC define'variables
0730
0740 PROC set'port'b
0750   POKE port'b'ddr,2
0760   POKE port'b,2
0770 ENDPROC set'port'b
0780
0790 PROC advance'pointer
0800   PRINT AT 10+position,2: " "
0810   IF position<4 THEN
0820     position:=position+1
0830   ELSE
0840     position:=2
0850   ENDIF
0860   PRINT AT 10+position,2: ">"
0870 ENDPROC advance'pointer
```

Index

XYZ

special characters