DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

COMMODORE 64

PHIL CORNES

OMNIBUS EDITION
BOOK ONE PLUS BOOK TWO
The first full-colour guide to easy programming

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

COMMODORE 64

THE DK SCREEN-SHOT PROGRAMMING SERIES

Never has there been a more urgent need for a series of well-produced, straightforward, practical guides to learning to use a computer. It is in response to this demand that The DK Screen-Shot Programming Series has been created. It is a completely new concept in the field of teach-yourself computing. And it is the first comprehensive library of highly illustrated, machine-specific, step-by-step programming manuals.

BOOKS ABOUT THE COMMODORE 64

This is Book Two in a series of unique step-by-step guides to programming the Commodore 64. Together with its companion volumes, it will build up into a self-contained teaching course that begins with the basic principles of programming, and progresses – via more sophisticated techniques and routines – to an advanced level.

ALSO AVAILABLE IN THIS SERIES

Step-by-Step Programming for the **ZX Spectrum**

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Acorn Electron**

Step-by-Step Programming for the **Apple IIe**

Step-by-Step Programming for the **IBM PCjr**

PHIL CORNES

After taking a B.A. in Mathematics and Computing, Phil Cornes has been involved in system development of computer-based education at British Telecom's National Training College. He has been a part-time technical author since 1978, and has become a regular contributor to personal computer magazines such as *Personal Computer World, Computing Today* and *Electronics Today International*. He has written a book and a large number of articles on programming and using the Commodore 64.

BOOK ONE


POWER
CLR HOME
INST DEL
RESTORE
RETURN
SHIFT
CRSR
CRSR
£
@
*
RESTORE
RETURN
SHIFT

DK

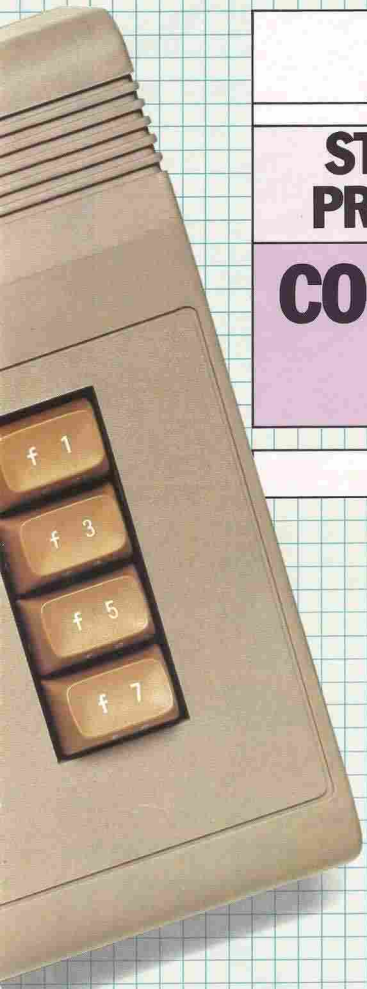
ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

COMMODORE 64

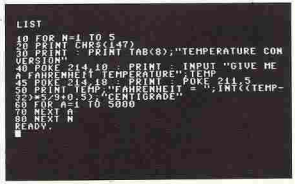
PHIL CORNES



GUILD PUBLISHING LONDON

BOOK ONE

CONTENTS



The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Designer Hugh Schermuly
Photography Vincent Oliver
Series Editor David Burnie
Series Art Editor Peter Luff
Managing Editor Alan Buckingham

First published in Great Britain in 1984 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

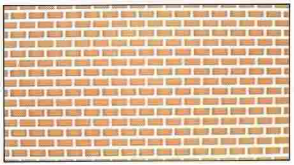
This edition published 1984 by Book Club Associates by arrangement with Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

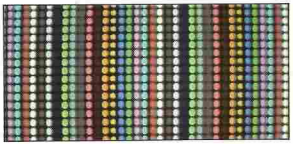
British Library Cataloguing in Publication Data

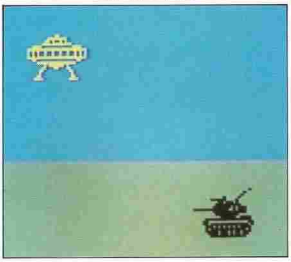
Cornes, Phil
Step-by-step programming for the Commodore 64. Bk. 1
1. Commodore 64 (Computer)——Programming
I. Title
001.64'2 QA76.8.C64

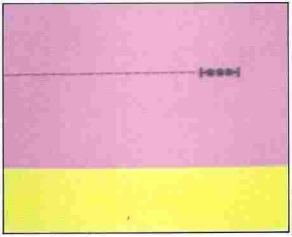
ISBN 0-86318-040-X

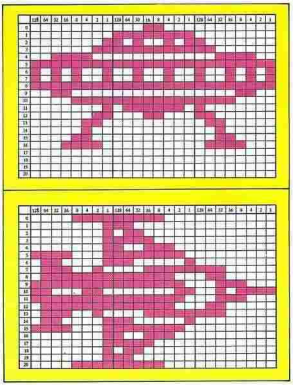
Typesetting by The Letter Box Company (Woking) Limited, Woking, Surrey, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona











THE COMMODORE 64

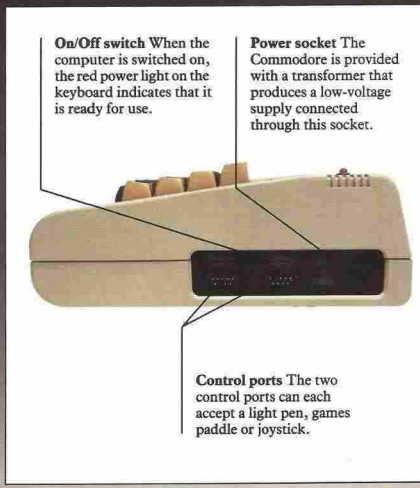
The Commodore 64 is an extremely versatile computer that provides many powerful facilities. These include sound synthesis on three channels, low- and high-resolution graphics in 16 colors, and high-resolution animation using small mobile object blocks called sprites. Once you have mastered the Commodore's simple dialect of the BASIC programming language, you will find that your Commodore will provide you with many hours of interest and entertainment.

Sockets and connectors

If you face the Commodore's keyboard, the sockets that enable you to connect it to other devices lie on the right-hand side and along the back panel of the computer. Turn the computer around and take a look at the back panel. From left to right the Commodore has a number of input and output sockets. First is the cartridge slot, into which pre-programmed Read Only Memory (ROM) program cartridges can be inserted. Cartridges are available for a range of uses like games, utilities and extra programming languages. The next three sockets are concerned with feeding television picture signals out to the screen. The first of these allows adjustment of the television channel used by the Commodore. The second is the UHF socket which provides a signal which can be fed straight into your television's aerial socket. If you have a Commodore video monitor, you will need to use the third socket which outputs an audio and color video signal. (All the photographs in this book were taken using displays produced on a Commodore monitor.)

Next in line is the serial port through which the Commodore printer and the Commodore disk drive unit are connected. This is followed by the cassette interface which is used to link up the special Commodore C2N digital cassette recorder. Finally, on the right is the user port. This is used mainly for interface circuits such as a modem (modulator-demodulator for telephone transmissions) or an RS232 communications cartridge.

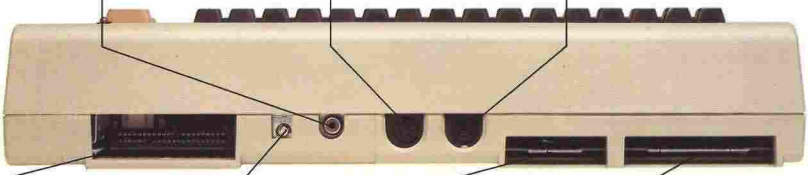
The list of connectors does not stop there. If you turn the computer around and look at the right-hand side panel you will see a switch and some more connecting sockets. Working from the front to the back, the first two connectors are the games or control ports. These accept analog or non-digital inputs from devices such as light pens, games paddles and joysticks. The signals are then converted into digital form to control programs. Next to the control ports is the main power on/off switch, and finally comes the power socket itself.



UHF socket This feeds the Commodore's sound and picture signals into a television aerial socket.

Audio/Video port A color video monitor can be connected through this socket, which provides high-quality sound and picture signals.

Serial port Interfaces for the standard serial printer and Commodore 1541 disk drive are provided via this socket.



Cartridge slot The Commodore can handle several extra languages and utility programs in Read Only Memory (ROM) cartridges. These are inserted into this slot.

Channel selector This allows some adjustment to be made to the television channel used by the computer. Normally no adjustment of this control is needed.

Cassette interface This connects the Commodore to the special C2N cassette recorder for program and data file storage.

User port The Commodore can be used to control various peripherals through the user port including a Centronics printer and other parallel devices.

INSIDE THE COMPUTER

The Commodore 64 is constructed on two circuit boards. The first carries the keyboard (see pages 10–11), while the second is the main board, which covers the whole floor of the case. The top shell of the case is designed to be removable, allowing the main board to be examined in detail.

The board contains the same basic elements which every personal computer uses – a microprocessor, various types of memory and input/output chips. The microprocessor (CPU) chip does all the computer's calculations and controls the activities of the rest of the machine. The Commodore 64 uses a type 6510 CPU. The computer's memory is divided into two main types – Random Access Memory (RAM) and Read Only Memory (ROM). RAM is also called "volatile memory" because its contents are lost when the power is interrupted. RAM is a temporary memory store; information is held there only while the power is on. ROM on the other hand is a permanent memory. The instructions it contains are not erased by the computer being turned off.

BASIC and machine code

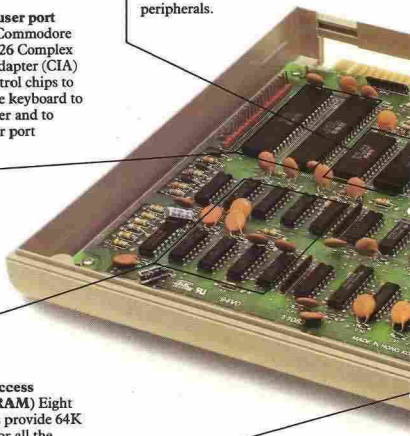
The computer's working languages are composed entirely of electronic pulses. The counting system that the computer's circuits use is based on only two numbers, 0 and 1, where 0 is represented by "off" (no pulse) and 1 by "on" (pulse present). This system is called binary code. In common with most personal computers, the Commodore deals with binary data in groups of eight binary digits (bits) at a time. Each group of eight bits (a bit is a single 0 or 1) is called a "byte". The data in each byte of memory represent a single character or symbol on the keyboard. The Commodore 64 contains 64K of RAM and 16K of ROM. The "K" stands for kilobyte (1K=1024 bytes).

The Commodore is most easily programmed with the "high-level" language BASIC. The BASIC on the Commodore is fairly compact, and it occupies only 8K of ROM. This means that some of the facilities of the machine have to be accessed in a fairly general way instead of by using specific BASIC keywords. Before the computer can act on the instructions you give it in BASIC it must first translate them into machine code, the language that the CPU understands. BASIC is therefore slower than machine code but has the advantage of being much easier to understand. When the computer is switched on, it automatically selects the program in the BASIC ROM. This program is called the BASIC interpreter.

The microchip command chain All the chips within the Commodore form an electronic chain of command, with the CPU performing all of the executive tasks. The rest of the chips, including the RAMs and ROMs, act as temporary or permanent information storage systems. These supply the CPU with the instructions it needs to carry out the computer's functions.

BASIC and KERNAL ROMs The BASIC ROM contains the instructions necessary to turn programs into a form that the computer's most important chip, the CPU, can understand. The KERNAL ROM provides instructions for communication with peripherals.

Keyboard/user port chips The Commodore uses two 6526 Complex Interface Adapter (CIA) parallel control chips to interface the keyboard to the computer and to provide user port facilities.



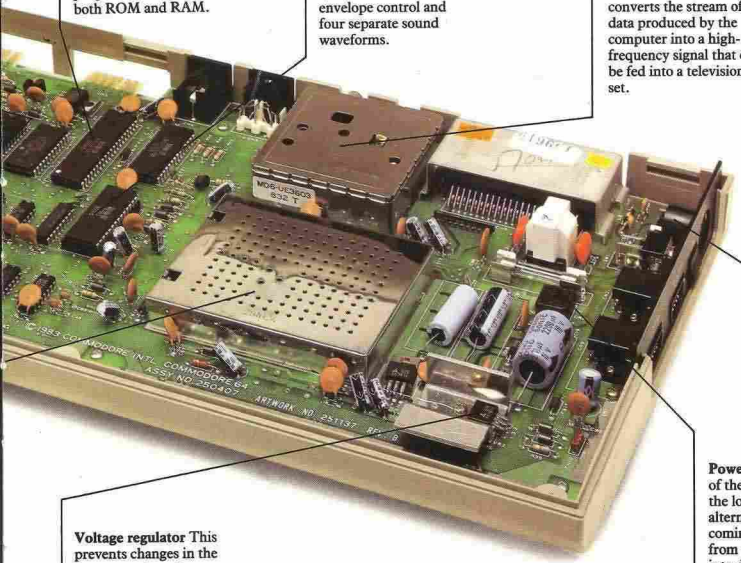
Random Access Memory (RAM) Eight RAM chips provide 64K of storage for all the programming information that the computer is given after being switched on.

Video Interface Circuit (VIC) The VIC chip controls all the Commodore's low- and high-resolution graphics, including color and sprites.

Central Processing Unit (CPU) The executive part of the computer. This microprocessor carries out all the calculations and controls the activities of the rest of the computer, using information and programs contained in both ROM and RAM.

Sound Interface Device (SID) The SID chip provides a complete three-channel sound synthesizer with sound envelope control and four separate sound waveforms.

Phase Alternation Line (PAL) encoder This converts the stream of data produced by the computer into a high-frequency signal that can be fed into a television set.



Power socket This is the Commodore's low-voltage input point.

Power supply This part of the computer converts the low-voltage alternating current (AC) coming into the computer from the transformer into direct current (DC) which can be used by the computer's circuitry.

Voltage regulator This prevents changes in the power supply voltage disrupting the activities of the computer.

THE COMMODORE KEYBOARD

The Commodore has a high-quality keyboard which is equally suitable for one-finger programming or fast touch-typing. It looks very similar to a typewriter keyboard, but the central block of ordinary letter and number keys is surrounded by a number of extra keys not found on a typewriter. The keys on the board can be split by function into three groups – the central block with numbers and letters on them, the surrounding dark-coloured keys, and the four light-colored keys to the right of the main block.

Character keys

When one of the central block of keys is pressed it produces a character on the screen. You can use these keys to type in words that the computer will recognize as commands, or information that you want the computer to use while it is running a program. You will notice that as well as having letters or numbers printed on them, these keys also have symbols or words printed on their front faces. The symbols are a set of graphics characters that can be made to appear on the screen, taking the place of the letters. The number keys control the colors, and can be used to change the colors of words and graphics as you type. Full descriptions of how to use these graphics characters and color controls are given later in this book.

Cursor and editing keys

On the right-hand side of the main block of keys there are four special keys that control the movement of the screen cursor and allow editing. The Commodore enables programs to be edited and corrected in a very flexible way. Any lines of a program to be edited or corrected are first listed on the screen, and then picked out by using the two keys labeled CRSR, which are in the bottom right-hand corner of the keyboard. The INST/DEL (INSerT/DELete) key, at the top right-hand corner of the keyboard, can then be used to insert and delete characters. The CLR/HOME key clears the screen and moves the cursor to the top left corner.

Function keys

If you press one of the four light brown keys to the right of the main block nothing appears to happen. This is because the Commodore function keys are designed to be used from within a program, and perform no action under normal circumstances.

RUN/STOP This key stops a program that has been set running, and shows where the program has been stopped.

CTRL This key has a similar function to the Commodore key; it allows access to the eight colors marked on the number keys 1–8. It can also be used with the 9 and 0 keys to produce "reversed" characters.

Commodore key The main function of this key is to allow access (with the number keys 1–8) to the second set of colors available on the Commodore. It is used with letter keys to produce any of a set of graphics characters on the screen.

Number keys As well as producing the numbers 0–9, these keys can also be used in conjunction with the CTRL and Commodore keys to give access to any of 16 colors.

RETURN The Commodore will not respond to most commands unless they are followed by pressing this key. It is roughly equivalent to a typewriter's carriage return.

RESTORE When used with the RUN/STOP key, this resets the computer to produce a clear screen.

CLR/HOME This is one of the Commodore's four cursor control keys. Pressing it on its own will make the cursor home to the top left corner of the screen. Pressing this key at the same time as the SHIFT key gives a method of clearing the screen from the keyboard.

INST/DEL This key is used for program editing. On its own it backspaces the cursor and deletes characters on the screen. Used with the SHIFT key it has the opposite effect and opens up a gap in a line and inserts spaces.

Space bar This works exactly like the space bar on an ordinary typewriter.

SHIFT and SHIFT LOCK These keys allow access to a set of graphics characters on the letter keys, and, when used in combination with the Commodore key, will change a display from one character set to the other.

Function keys These four keys can be used with the SHIFT key to provide eight key combinations that can be detected within a program and used to provide controlled functions.

Cursor control keys These two keys, used in conjunction with the SHIFT key, allow the cursor to be moved in any direction around the screen. This facility is needed when you are editing parts of a program.

SETTING UP

Setting up the Commodore is a two-part operation. First, you need to connect the computer up to its peripherals, and then you need to adjust your television or monitor to get the best results on the screen. The first part is easy – the second can take a little longer.

The system as supplied comprises the main keyboard unit, which also contains all of the computer's electronics, a power supply transformer and an aerial lead to connect the computer to a television set.

The power supply transformer has two leads. The DIN plug should be connected with the computer power socket which is on the right-hand side of the case. The other lead is the power lead which should be plugged into a wall socket. Now, if you are using a television, unplug the aerial lead and replace it with the Commodore television lead. The other end of this should be plugged into the UHF TV socket on the computer's rear panel. If you are using a Commodore monitor, connect this through the audio/video port.

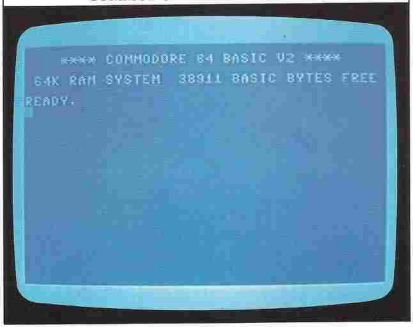
When you switch on the computer at the power switch, the red power indicator light at the top right of the keyboard should come on.

If you are using a television, switch on and select a spare channel. (It helps if you can keep a channel permanently allocated to the computer.)

Tuning in

When the computer is switched on, it produces a screen signal that you should be able to receive on your television set. After some experimentation with the tuning controls, you should see this display – or one similar – on your screen:

COMMODORE SCREEN DISPLAY



If you can't get this display at all, you probably have not connected the computer up properly. Check all the leads and connections again.

If you are using a color television, you should see a color display like the one shown. If the colors seem very different, try fine-tuning the set. When a television is slightly out of tune, it may produce a sharp black-and-white picture with no indication that color signals are being produced. If you have a monitor, it will be tuned already. Adjustment of the controls will enable you to select the best color settings.

Sometimes with a television, the display will not be centered properly. You can correct this by using the normal horizontal and vertical hold adjustment controls. Remember also that the Commodore has no built-in loudspeaker. Instead it uses the loudspeaker in the television or monitor. When you come to writing programs that produce sound, check that the volume control on your set is turned up so that any sounds which the computer produces are audible.

You will notice when you have set up the computer that its initial display contains the line:

38911 BASIC BYTES FREE

This number tells you how much memory space there is inside the computer that can be used to store BASIC programs and the data that these programs operate on. A byte of memory will hold one character – a letter or symbol. As you begin programming, memory will start being used (although the screen display will not register this). Because your programs are only held in memory for as long as the power is on, the computer will always have the same number of bytes free every time you switch on.

Commodore color combinations

In addition to the light and dark blue colors seen on the initial display, the Commodore is capable of producing many other colors. There are 16 colors in total, giving you a very large number of color combinations that can be displayed on the screen. As you will soon find, some of these – particularly combinations of strong colors – look very good. Combinations which use just the pale colors often prove less successful. Some color combinations may produce "fringing", or shadowing which makes any text on the screen difficult to read. This is not a result of faulty tuning, and the best way to get around the problem is to change to other colors. The color combinations in this book will give you an idea of some colors which go together well.

Connecting peripherals

Peripherals are items of equipment such as cassette recorders, disk drives and printers that you can connect to your machine. Apart from display screens, there are many different peripherals available for the

Commodore. The one you are likely to use most often is a Commodore cassette recorder or Commodore disk drive, both of which are used to store and play back programs. Commodore produces cassette recorders and disk drives that are specifically designed for use with your computer. A disk drive performs just the same function as a cassette recorder but, as well as saving and loading programs faster, it has almost immediate access to any program on a disk without having to carry out a search from the beginning, as a cassette recorder does. This reduction in access time is very useful if you want to record and play back programs frequently. Using a cassette recorder and a disk drive is covered in detail on page 58.

In general, when you are making plug-socket connections, never force a plug into a socket if you feel any resistance. You may be trying to push a plug into the wrong socket. If not, check that there is no debris inside the socket, and make sure that none of the pins is bent. When you want to disconnect a plug, never pull it out by the lead – you may put enough strain on a wire or connection inside the plug to break it.

Coloring listings and displays The Commodore's light and dark blue screen display can be switched to other colors either directly, using the keyboard color controls, or indirectly, by programming color changes. Program listings will normally appear in the two shades of blue.

In order to make the programs in this book as clear as possible, the computer has been programmed to show the listings in white on black (the method for doing this is described on page 32). This will have no effect on your programs; you can enter them in any color you want.

STARTING OFF

Having set up your Commodore, you may already have given in to the temptation to tap a few keys to see what happens. If not, try it – you can't do any damage. In most cases, pressing a key causes the symbol on the key-top to appear on the screen.

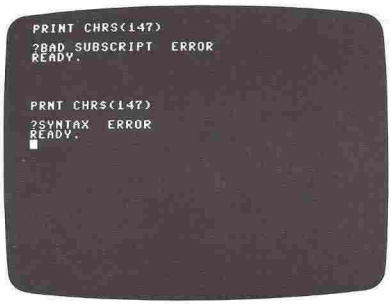
But having successfully got the computer to display something, you will then want to know how to remove it. The simplest method is to hold SHIFT and press CLR (the second key from the right on the top row). This erases anything on the screen. To reset the computer so that any commands that you may have given it are lost from its memory it is easy to turn the computer off and on again. But be careful, because this can irretrievably erase a carefully written program.

Another way of clearing the screen is to type PRINT CHR$(147) then press the RETURN key.

Giving exactly the right instructions

It is important to remember that the computer will only obey instructions that are exactly correct. If you type in PRINT CHR$(147) and RETURN the screen will clear. But if you type in, for example, PRINT CHRS(147), you will just get an error message, one of many that the computer has stored in its permanent memory. This is because the computer does not understand what you have typed. This is how the computer responds to incorrect instructions:

INCORRECT COMMAND ERROR MESSAGES



So if during the following pages your computer refuses to obey your instructions, make sure you have not given it a command that it cannot recognize.

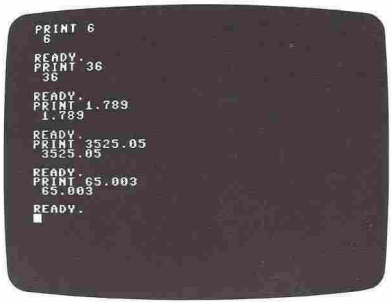
Starting to PRINT on the screen

Now clear the screen and type in this line:

PRINT 6

If you press the RETURN key after this, the number 6 will appear on the next line of the screen. The computer has responded to your command. PRINT has nothing to do with ink and paper – it just tells the computer to display something on the television screen. Try using the same command in the same way with other numbers. It doesn't matter whether you leave a space between PRINT and the number for neatness, or not. The computer can still understand the instruction:

PRINT WITH NUMBERS

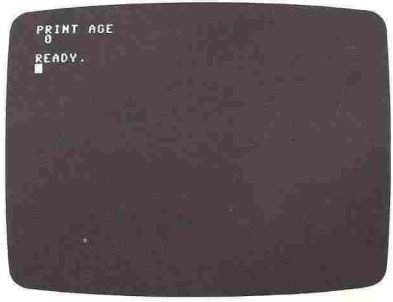


But now clear the screen with SHIFT and CLR again and then type in:

PRINT AGE

The computer responds by displaying a zero and not the word AGE as you might have expected:

PRINT WITH WORDS



What has happened is that the computer has been hunting in its memory for a "variable" called AGE, and

because it cannot find one, it creates it and gives it the value zero. This value is what you then see PRINTed on the screen.

What is a variable?

Without clearing the screen now type in the instruction in a different way:

PRINT "AGE"

This time the computer makes the correct response – it PRINTs AGE on the next line of the screen. You have just discovered that to the computer AGE (on its own) and "AGE" (inside quotation marks) mean two totally different things. The computer treats any letter or group of letters on their own as a variable. A variable is simply a label identifying a number stored in the computer's memory.

Now you should be able to see why PRINT AGE had unexpected results. The computer read AGE not as a word but as a label for a slot in its memory. It looked for a number called AGE, but because you haven't used the computer's memory yet, it couldn't find one.

Numeric and string variables

To make PRINT AGE more comprehensible to the computer, give AGE a value. Try this (press the RETURN key at the end of each line):

USING LET

```
PRINT AGE
 0
READY.

PRINT "AGE"
AGE
READY.

LET AGE=14

READY.
PRINT AGE
 14
READY.
```

Now AGE is labeling something, the number 14. LET is a command which gives a label and a value to a slot in the memory. Every time you ask the computer to PRINT AGE, it will display the last value entered. "AGE" is called a string, while AGE, which always represents a number, is called a numeric variable.

The computer displays everything put inside quotation marks exactly as you type it. Try it. You can use any characters on the keyboard – letters, numbers, mathematical symbols and punctuation marks – as long as the PRINT command doesn't go over two lines:

PRINTING STRINGS

```
PRINT "SCORE"
SCORE

READY.
PRINT "COMMODORE"
COMMODORE

READY.
PRINT "THE NEXT FLIGHT LEAVES AT 13.45"
THE NEXT FLIGHT LEAVES AT 13.45

READY.
```

The characters between the quotation marks make up a string. In the same way as a number is stored in the computer and labeled by a numeric variable, a string is stored and labeled by a string variable. String variables always end in a dollar sign. A$, NUMBER$, PRICE$ and CITY$ are all string variables. In the line:

LET CITY$="NEW YORK"

for example, CITY$ is a string variable and NEW YORK is the string it labels. Clear the screen again and then type the above line on the keyboard. If you type:

PRINT CITY$

the computer will then reveal the contents of the string variable CITY$. As with numeric variables, the command LET allows you to put a string into the computer's memory. Again, the computer will remember only the last version of the string labeled by a particular variable in its memory, so you can change CITY$ as often as you like:

CHANGING A STRING

```
LET CITY$="NEW YORK"

READY.
PRINT CITY$
NEW YORK

READY.
LET CITY$="LONDON"

READY.
PRINT CITY$
LONDON

READY.
LET CITY$="CHICAGO, SAN FRANCISCO"

READY.
PRINT CITY$
CHICAGO, SAN FRANCISCO

READY.
```

KEYING IN COLOR

One of the Commodore 64's outstanding features is its ability to produce dazzling color. The machine can display 16 colors and these can all be accessed directly from the keyboard, that is, without a program being needed. On these two pages, you will learn how to change the color of text on the screen. As you will see, there are quite a number of different ways of doing this.

When you switch the computer on, the display is blue and light blue, or cyan. Many of the illustrations in this book show white text on a black background, which is easier to read than the normal display. You will see how to produce white text in a moment, but if you also want to change the screen to black, type in:

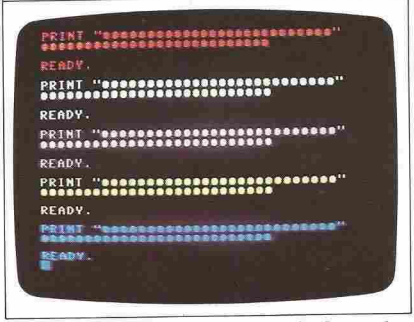
```
POKE 53280,0:POKE 53281,0
```

and then press RETURN. You will find out how these commands work on pages 32–33.

Changing colors from the keyboard

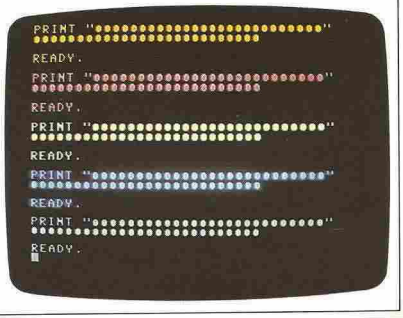
If you look at the Commodore keyboard you will see 8 colors lettered onto the fronts of number keys 1 to 8. These keys are used in conjunction with the CTRL key (furthest left on the second row) to select 8 of the 16 available colors. Try a few simple exercises. Press and hold the CTRL key, and then press the key marked BLK (the 1 key). You will notice that the flashing cursor on the screen has changed to black and that anything you now type also appears in black. Similarly, if any other number key is pressed with the CTRL key then the cursor color will change to that shown on the number key. This screen shows a series of colors selected with the CTRL key (the solid circle is selected by pressing SHIFT and Q):

CTRL KEY COLORS



The other colors that are available on the Commodore are selected in the same way, but with the key marked C= (the Commodore key) instead of CTRL. The next screen shows some of these colors, and the table following it shows you how to access them:

COMMODORE KEY COLORS



COMMODORE TEXT COLORS

The text colors produced by the number keys are selected in combination with the CTRL or Commodore (C=) keys.

Number key	Color produced with CTRL key	with C= key
1	Black	Orange
2	White	Brown
3	Red	Light red
4	Cyan	Dark gray
5	Purple	Medium gray
6	Green	Light green
7	Blue	Light blue
8	Yellow	Light gray

Changing colors with control symbols

As well as using the Commodore's colors directly, you can also put special control symbols in a string using the command LET. Doing this prevents the color changes from occurring until you tell the computer to PRINT

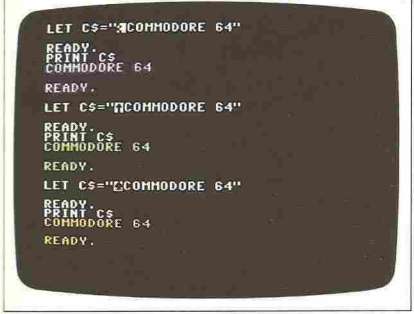
COLOR CONTROL SYMBOLS

Each of the Commodore's 16 colors can be keyed in as a string to produce a color control symbol on the screen.

Color	Selector keys	Control symbol	Color	Selector keys	Control symbol
Black	CTRL 1		Orange	C=1	
White	CTRL 2		Brown	C=2	
Red	CTRL 3		Light red	C=3	
Cyan	CTRL 4		Dark gray	C=4	
Purple	CTRL 5		Medium gray	C=5	
Green	CTRL 6		Light green	C=6	
Blue	CTRL 7		Light blue	C=7	
Yellow	CTRL 8		Light gray	C=8	

the string, a feature which can be used in a program. This is what the process looks like on the screen:

COLORING WITH CONTROL SYMBOLS



The keys used to enter the special control symbols are the same ones you have been using in the direct mode – the number keys with CTRL or C=. But as you can see, what appears within the string with this method bears no relation to the color chosen. Each control symbol is just one of the many graphics characters that are permanently programmed into the Commodore. Although it is quite easy to key in color symbols, copying them in from a listing can be more difficult. In order to avoid any confusion with color changes in the programs in this book, we shall often use a third way of changing colors, one which uses numeric "character codes".

Changing colors with character codes

Inside the Commodore a special code called ASCII (American Standard Code for Information Interchange) is used to represent a set of characters that can be PRINTed on the screen. So, for example, the capital letter A in ASCII is represented by the number 65, B by the number 66 and so on. A complete list of these codes is given on page 61. But as well as representing all of the PRINTable characters, ASCII codes also activate operations such as cursor movement, line changing and character color controls. In Commodore BASIC there is a special keyword that can be used with PRINT to produce a character or operation if its ASCII code is given. This is CHR$. You have come across this already in PRINT CHR$(147), which clears the screen.

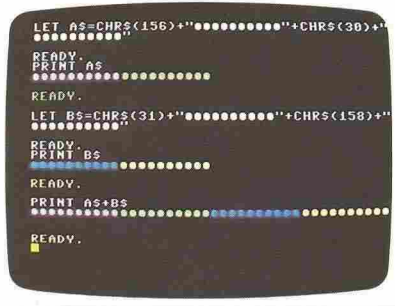
As you can see from the table on this page, the ASCII color codes do not form a continuous series. However, despite these drawbacks, with CHR$ there is less chance of errors creeping into programs as you copy them, because numbers are much easier to deal with than graphics symbols. This is the text color-changing method that will be used most often in this book. If you want to use other color control methods, the programs will work just as well. Here are some CHR$ color changes:

COLORING WITH **CHR$** CODES

Using CHR$ in strings

In the same way as you can use LET to type color control codes into a string, so you can also type in CHR$ codes. Again, they only work after PRINT:

COLORING WITH **LET** AND **CHR$**



ASCII COLOR CODES

The CHR$ command can be used with ASCII codes directly or in programs to produce the colors that are accessed with the CTRL or Commodore keys.

Color	ASCII code	Color	ASCII code
Black	CHR$(144)	Orange	CHR$(129)
White	CHR$(5)	Brown	CHR$(149)
Red	CHR$(28)	Light red	CHR$(150)
Cyan	CHR$(159)	Dark gray	CHR$(151)
Purple	CHR$(156)	Medium gray	CHR$(152)
Green	CHR$(30)	Light green	CHR$(153)
Blue	CHR$(31)	Light blue	CHR$(154)
Yellow	CHR$(158)	Light gray	CHR$(155)
Reverse on	CHR$(18)	Reverse off	CHR$(146)

COMPUTER CALCULATIONS

The PRINT command is not limited just to displaying characters on the screen. You can also use it in conjunction with the four mathematical functions – addition, subtraction, multiplication and division – to perform calculations that you can follow on your screen.

Let's take addition first. The plus sign is on the fifth key in from the right on the top row. To add two numbers together, simply use PRINT followed by the calculation. Subtraction is carried out in the same way. The minus sign, which doubles as a hyphen when used in text, is next to the plus key. The screen below shows simple additions and subtractions, the bottom screen shows multiplications and divisions:

ADDING AND SUBTRACTING

```
PRINT 63.8+55
 118.8
READY.
PRINT 566+155
 721
READY.
PRINT 61.5-32.02
 29.48
READY.
PRINT 1167-622
 545
READY.
```

MULTIPLYING AND DIVIDING

```
PRINT 9*11
 99
READY.
PRINT 6.25*24
 150
READY.
PRINT 100/4
 25
READY.
PRINT 84/4.5
 18.6666667
READY.
```

Multiplication is carried out, not with the familiar × but with an asterisk, *. The asterisk is just under the minus key. Division uses the oblique stroke, /, next to the right SHIFT key. In 24/8, for example, the left-hand number is divided by the right-hand number. As you can see in the second screen, when a division produces a recurring result, as in 84/4.5, the computer gives a nine-figure answer.

Calculating exponents and square roots

In addition to these familiar math functions, you can multiply a figure by itself a specified number of times (called exponentiation), and find out square roots. For example, 2^3 is equivalent to 2 multiplied by itself three times, in other words 8. The keyboard cannot produce superscripts like the 3 in 2^3, so you have to use the up arrow (↑) symbol below the CLR key. 2^3 is keyed into the computer as 2 ↑ 3. Here are some examples:

EXPONENTS

```
PRINT 2↑3
 8
READY.
PRINT 8↑2
 64
READY.
PRINT 2↑5
 32
READY.
PRINT 10↑6
 1000000
READY.
```

The computer also allows you to find out the square root of a number. This time there isn't a single key that carries out the calculation; instead you have to type in the command like this:

PRINT SQR(2)

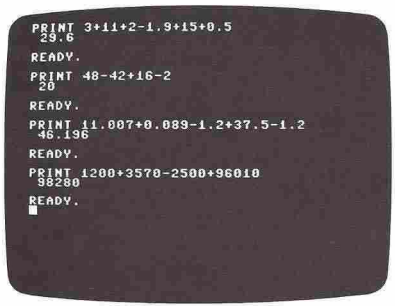
Make sure that you use the round brackets on the number keys 8 and 9, and not the square brackets to the left of the RETURN key. Because the brackets are the upper of the two symbols on the key-tops the SHIFT key must be pressed with the bracket keys in order to select the upper characters. When you press RETURN after keying in this line, the computer will PRINT out the answer. However, if you try this command with a minus number, the computer will produce an error message to let you know that you have asked for a mathematical impossibility.

How to specify a sequence of calculations

You can carry out a number of different calculations

using the same PRINT command. Try it with just addition and subtraction first. You will discover that the machine's memory seems inexhaustible:

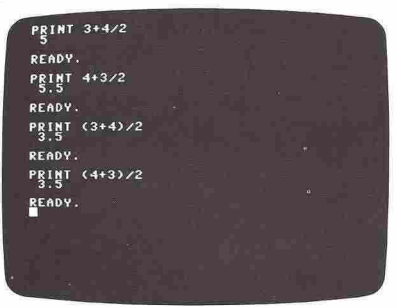
MULTIPLE CALCULATIONS



You can enter the figures for each calculation in any order at all, and the result will be the same.

However, when you add multiplication and division to the chain of calculations, apparently odd things may happen. Look at the next screen, and try the calculations for yourself. Say you want to add two numbers together and divide the result by 2. The order in which the numbers are added should not make any difference to the result, but it appears to do so:

DIFFERENT RESULTS FROM THE SAME CALCULATION



If 3+4 is exactly the same as 4+3, then why should the subsequent division by 2 produce different answers? The reason is that the computer does not necessarily carry out calculations in the order you key them in on the screen. It performs exponentiation first, then multiplication and division, and finally addition and subtraction. So in PRINT 3+4/2, 4 is divided by 2 before 3 is added to the result. But in PRINT 4+3/2, 3 is divided by 2 before 4 is added to the result.

The problem you set the computer was to add two numbers together and divide the result by 2. Neither of these examples does this. But you can change the order in which the computer performs calculations by using pairs of round brackets again, as in the final two examples on the screen. Here, the addition is carried out first and then the result is divided by 2.

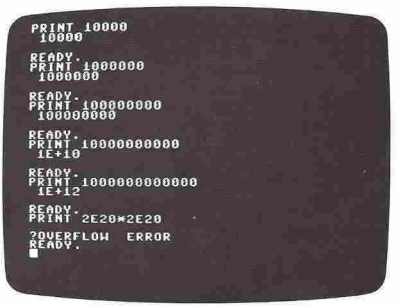
What are the computer's limits?

There are limits to the numbers that the computer can handle, and these limits take two forms – size and accuracy. The size limitation is unlikely to inconvenience you. Numbers with a decimal point can have any value in the range 1×10^{38} (1 followed by 38 zeros) to -1×10^{38} (-1 followed by 38 zeros). Integers (whole numbers) can have any value from 32,767 to $-32{,}768$.

The accuracy of these two types of numbers is also handled in a slightly different way in each case. Although numbers with decimals can be up to 1 followed by 38 zeros, the computer only memorizes the first nine digits – the rest it sets at zero. This nine-figure accuracy is adequate for most applications. Whole numbers are stored with complete accuracy, so that within its range the computer remembers whole numbers with an accuracy better than 5 thousandths of 1 percent!

You may come across another of the computer's quirks when dealing with very big numbers. The Commodore does not display them in the way in which you would type them on the keyboard. For example, PRINT 1000000000000 produces 1E+12 on the screen (the E stands for exponent). This is simply a shorthand way of displaying 1×10^{12} or 1 followed by 12 zeros, the number you keyed in. Try entering some large numbers and calculations using the PRINT command and see the way the computer responds to them:

PRINTING LARGE NUMBERS



WRITING YOUR FIRST PROGRAM

So far, you have given your Commodore commands to which it has responded immediately. These commands have been very simple – in many cases it would have been quicker not to have used the computer at all. However, commands on their own are not computer programs. The computer reads each command, carries it out, and then forgets it. A program on the other hand is an orderly list of instructions which the computer can store in its memory. It can carry them out as and when you wish so that a long and complicated set of instructions can be repeated at the touch of a button.

From single commands to program lines

Having found a task that you want your Commodore to carry out, the next job is to write the program in steps that the computer can understand. The Commodore, like most personal computers, uses a computer language called BASIC (Beginners' All-purpose Symbolic Instruction Code). BASIC is an example of a high-level language, a language composed of words and symbols with which you, the user, are already familiar. It is, therefore, one of the easiest computer programming languages to learn – as you are about to find out!

The essence of a program is that it is stored in the computer's memory. The commands that you have keyed into your computer so far can be turned into programs simply by adding line numbers. Here is a simple six-line program:

SQUARE CALCULATION PROGRAM

```
10 PRINT CHR$(147)
20 LET X=150
30 LET TEXT$=" SQUARED = "
40 PRINT : PRINT " **********************
**********"
50 PRINT : PRINT TAB(8);X;TEXT$;X↑2
60 PRINT : PRINT " **********************
**********"
```

As you key the program in, you will notice that the commands are not now carried out as soon as you press the RETURN key. Instead the program is safely stored in the computer's memory. It remains only to run it – by typing RUN, and then RETURN – and to see what it then produces on the screen.

How program lines are numbered

You may be wondering why the lines are numbered in tens. When you are writing and testing programs, you will frequently want to add extra lines. If the existing lines are numbered 1, 2, 3, 4, and so on, there is nowhere to put the new lines so that the computer carries them out in the right order. In the previous program, there is room to add extra lines numbered 0–9, 11–19 and so on if necessary.

The program is still in memory, so to try the next one switch off and on to reset the computer and erase the old program. Then see what the following produces when you key in the program and then type RUN:

SCREEN DISPLAY PROGRAM

```
10 REM SCREEN PRINT
20 PRINT CHR$(147)
30 PRINT TAB(9);"%";TAB(27);"%"
40 PRINT TAB(9);"%";TAB(12);"DEMONSTRATI
ON";TAB(27);"%"
50 PRINT TAB(9);"%";TAB(11);"SCREEN  DIS
PLAY";TAB(27);"%"
70 PRINT TAB(9);"%";TAB(27);"%"
80 PRINT TAB(9);"%%%%%%%%%%%%%%%%%%%%%%%"
```

Taking it from the top, what is a REM? REM is short for REMark. The computer doesn't do anything with a REM statement other than store it with the rest of the program. But it's a useful device for titling or labeling programs or lines, so that you can find them quickly. As your programming ability grows, you will find REM lines very valuable for reminding you how a particular program works. Other people will also be able to follow your programs more easily if there are periodical REM statements to explain what you are doing or how the program is structured.

PRINT CHR$(147) you have come across already. It's a quick way of taking all the old unwanted information off the screen. Lines 30 to 80 PRINT % symbols in a frame around the text. They use the command TAB to position the symbols along each line. There are 40 character spaces per line, numbered from 0 to 39. TAB works just like the tab space on a typewriter, so that PRINT TAB(9) means that PRINTing will begin at position number 9, instead of at the left-hand edge of the screen. Lines 50 and 60 PRINT the text itself in the middle of the frame.

Why punctuation is important

The next program uses the techniques demonstrated on pages 18–19. Switch off the power again for a second or two before keying it in:

CALCULATIONS PROGRAM

```
10 PRINT CHR$(147)
20 PRINT : PRINT TAB(8);"43/3 =";43/3
30 PRINT : PRINT TAB(8);"3*18 =";3*18
40 PRINT : PRINT TAB(8);"6.25*4 =";6.25*
4
50 PRINT : PRINT TAB(8);"179*9.8 =";179*
9.8
60 PRINT : PRINT TAB(8);"1002/28.7+3 =";
1002/28.7+3
70 PRINT : PRINT TAB(8);"100*9/5+32 ="10
0*9/5+32
```

```
          43/3 = 14.3333333
          3*18 = 54
          6.25*4 = 25
          179*9.8 = 1754.2
          1002/28.7+3 = 37.912892
          100*9/5+32 = 212

READY.
```

In each of the six calculations, the screen shows the calculation and the result. The PRINT command at the beginning of each program line produces a space (blank line) before the following calculation is displayed on the next line down.

The semi-colon is very important. It ensures that the result is shown on the same line as, and immediately following, the details of the calculation. Try the same program with a comma, a colon and nothing at all instead of the semi-colon. You'll quickly realize how important punctuation is in computer programming. Correct spacing is also vital if you want to produce a readable display when the computer PRINTs numbers and strings following each other.

The following program, which again combines some calculations with PRINT, shows how to plan a display. It works out some conversions on screen:

CONVERSIONS PROGRAM

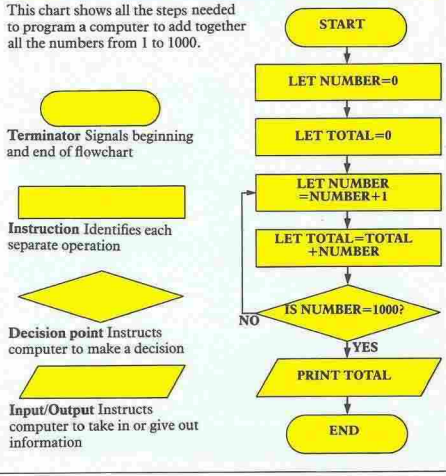
```
LIST

10 PRINT CHR$(147)
20 PRINT "CONVERSIONS"
30 PRINT
40 PRINT
50 PRINT "1 FOOT =";12*2.54;" CENTIMETER
S"
60 PRINT
70 PRINT "30 CENTIGRADE =";(30*9/5)+32;"
 FAHRENHEIT"
80 PRINT
90 PRINT "10 KILOMETERS =";10*5/8;" MILE
S"
100 PRINT
110 PRINT "1 YEAR =";365*24*60*60;" SECO
NDS"
READY.
```

How to write a flowchart

If a program is to RUN properly, it must carry out the correct operations in the right order. Drawing a flowchart is a useful way of outlining the steps involved in making the computer perform a task. This flowchart shows how to plan a program to add up all the numbers from 1 to 1000. Each shape is a separate operation, and the arrows connecting the shapes show the path that the program is to follow. NUMBER and TOTAL can be entered in a program as numeric variables. This program contains two features which you will encounter later – a program "loop" and a program "decision point", which determines how many times the loop will be carried out.

DRAWING A FLOWCHART

This chart shows all the steps needed to program a computer to add together all the numbers from 1 to 1000.



Terminator Signals beginning and end of flowchart

Instruction Identifies each separate operation

Decision point Instructs computer to make a decision

Input/Output Instructs computer to take in or give out information

DISPLAYING PROGRAM LISTINGS

As you start writing programs, you will often want to refer back to check on something or perhaps alter it in some way. In order to do that, you must be able to see the program on the screen again after it has been RUN. The Commodore allows you to look at anything you have stored in its memory. In this case, you want to look at the "program listing" – the program as you typed it in.

If you've just switched the computer on again after a short break, type in a program from a previous page and RUN it to make sure that it's OK. The BASIC command LIST is used to call up your program onto the screen from the part of the memory where it is currently stored. Here it is used with the first program on page 20. The program has been keyed in, and then followed by LIST:

REPEATING A PROGRAM WITH LIST

```
10 PRINT CHR$(147)
20 LET X=150
30 LET TEXT$=" SQUARED = "
40 PRINT : PRINT " ********************
**********"
50 PRINT : PRINT TAB(8);X;TEXT$;X*2
60 PRINT : PRINT " ********************
**********"

LIST

10 PRINT CHR$(147)
20 LET X=150
30 LET TEXT$=" SQUARED = "
40 PRINT : PRINT " ********************
**********"
50 PRINT : PRINT TAB(8);X;TEXT$;X*2
60 PRINT : PRINT " ********************
**********"
READY.
```

Every time you press the RETURN key after writing a program line, the line is stored in the RAM. The program LISTing is an exact copy of all the lines that the RAM currently holds. The program is not permanently transferred from the RAM to the screen by LIST – it's still held in the memory.

Moving around a LIST

LIST's capabilities don't end there. Key in the program shown on the next screen. It incorporates a technique that you haven't covered yet, but that's not important. (Incidentally, if you want to RUN this program, press the RETURN key after you type in your name.) Then, to display the whole program listing again, type LIST. Say you want to look at the first line of the program only. You can do that by typing LIST 10. Perhaps you only need to see a few lines of the whole program. Say you want to look at lines 20 to 40. Try typing LIST 20–40. You can see the results on the second screen:

"OPERATOR" PROGRAM

```
10 PRINT CHR$(147)
20 PRINT : PRINT TAB(5);"----------
----------"
30 PRINT : PRINT TAB(7);"WHAT IS YOUR NA
ME"; : INPUT NAME$
40 PRINT : PRINT TAB(2);"COMMODORE 64 -
PROGRAMMED BY ";NAME$
50 PRINT : PRINT TAB(5);"----------
----------"
```

PARTIAL LISTING

```
10 PRINT CHR$(147)
20 PRINT : PRINT TAB(5);"----------
----------"
30 PRINT : PRINT TAB(7);"WHAT IS YOUR NA
ME"; : INPUT NAME$
40 PRINT : PRINT TAB(2);"COMMODORE 64 -
PROGRAMMED BY ";NAME$
50 PRINT : PRINT TAB(5);"----------
----------"

LIST 20-40

20 PRINT : PRINT TAB(5);"----------
----------"
30 PRINT : PRINT TAB(7);"WHAT IS YOUR NA
ME"; : INPUT NAME$
40 PRINT : PRINT TAB(2);"COMMODORE 64 -
PROGRAMMED BY ";NAME$
READY.
```

LIST also lets you look at everything up to a certain line number and everything from a certain line number to the end of the program. Type in LIST –50 and LIST 20– and watch the effect in each case.

If you want to search a large block of program for a particular line or lines the best way is to use LIST with no line numbers which will cause the whole program to "scroll" up the screen. The program will scroll rather quickly but when you are approaching the section of the listing containing the line you are trying to find you can slow the scrolling down by pressing and holding the CTRL key. This will allow you time to identify the required line. When you have found it, the LIST command can be aborted by pressing the STOP key. You can then LIST the specific line so that it can be examined. If necessary, it can then be edited using the techniques that are explained on pages 24-25.

Using NEW to delete a program

Imagine you are starting a new program. Clear the screen with SHIFT and CLR and then type in the first line:

```
10 PRINT "SPACE PROBE PROGRAM"
```

and RUN it. Something odd happens. The last program is still in memory. The computer carries out your new line 10, but then goes on to RUN the old program, because you haven't erased it.

Up to now you have been switching the machine off and on before entering a new program, but there are better ways of getting rid of old programs. One of them is to use the BASIC keyword NEW. Type NEW, press the RETURN key, then re-key in the new line 10. This time the old program appears to have gone for good. If you try to LIST any program after NEW, you will find that all its lines will have disappeared:

USING NEW

```
LIST
10 PRINT CHR$(147)
20 PRINT : PRINT TAB(5);"---------------
--------------"
30 PRINT : PRINT TAB(7);"WHAT IS YOUR NA
ME"; : INPUT NAME$
40 PRINT : PRINT TAB(2);"COMMODORE 64 -
PROGRAMMED BY ";NAME$
50 PRINT : PRINT TAB(5);"---------------
--------------"
READY.

NEW
READY.
LIST
READY.
```

Before using NEW, you should always be quite sure that you want to erase the program currently in memory. There is a way for getting a program back after NEW (this is shown on page 57), but it is quite a cumbersome process. In addition, it only works if you have not begun to key in another program.

RUNning a program segment

Programs may be RUN partially, either with RUN followed by a line number, or with the command GOTO. You may be having trouble getting part of a program to RUN properly. In a short program, it's just as convenient to RUN the whole program as it is to RUN only a part of it. But what if the troublesome part that you want to experiment with and RUN over and over again comes near the end of long program? It soon becomes tiresome and time-wasting to have to watch the first five minutes or so of the program unfold on the screen every time you want to check on the suspect part near the end. Suppose that in the square calculation program on page 20, you only want to check that it RUNs from line 60 onward. Instead of using RUN, try typing RUN 60 and GOTO 60 and see what happens:

PARTIALLY RUN PROGRAM

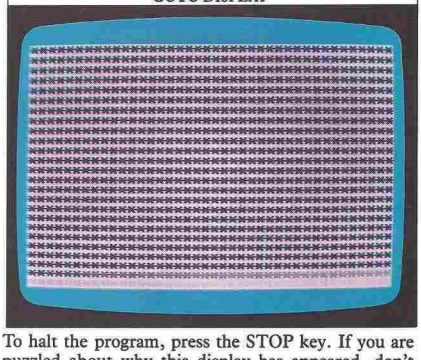
```
LIST
10 PRINT CHR$(147)
20 LET X=150
30 LET TEXT$=" SQUARED = "
40 PRINT : PRINT " ***************
**************"
50 PRINT : PRINT TAB(8);X;TEXT$;X↑2
60 PRINT : PRINT " ***************
**************"
READY.
RUN 60

 ***********************************

READY.

GOTO 60

 ***********************************

READY.
```

GOTO is one of the simplest and most useful commands in the BASIC language. Used on its own – without a program line in front of it – GOTO makes the computer go straight to a specified line and then RUN the program from that point. But when GOTO is actually part of a program, the results then become very interesting. You can get an idea of what this command can do simply by keying in two program lines:

```
10 PRINT "*";
20 GOTO 10
```

When you RUN it, your screen should look similar to this, depending on which colors you have selected:

GOTO DISPLAY



To halt the program, press the STOP key. If you are puzzled about why this display has appeared, don't worry. We will be returning to GOTO soon after you have mastered a few more BASIC keywords.

CORRECTING MISTAKES

Computer programming is one pastime in which mistakes are unavoidable. Programs very rarely work satisfactorily first time, and the longer they are, the more difficult it is to get them right. It's important to realize that making mistakes and correcting them is often an interesting part of program development. So, don't ignore, hide or gloss over your mistakes – they are an invaluable aid to learning how to get things right.

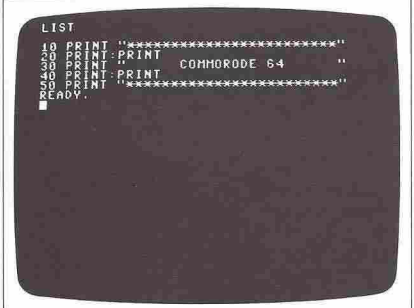
For instance, in a computer program you cannot alter punctuation without completely changing the sense of what you have written. As you saw on page 21, it will have drastic results. To the computer, punctuation means something very precise, and if you get it wrong, a program may not work.

You can change a line in a program in two ways. First, you can retype the line and press RETURN. The new version automatically replaces the old one in the computer's memory. However, if there is very little wrong with a line, especially if it is a long line, it's a waste of time to completely retype it. The alternative way of making a change in this case is to use the cursor keys to edit on screen.

Editing on the screen

Editing involves using the two CRSR keys with arrows printed on them and the INST/DEL key. Here is a program that needs editing:

PROGRAM BEFORE EDITING



To correct the spelling in line 30 to read:

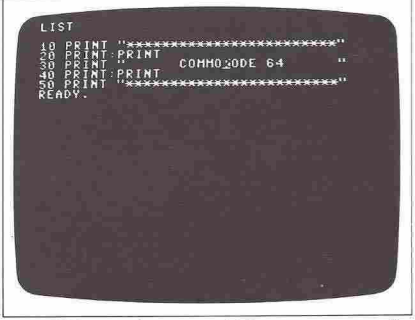
30 PRINT " COMMODORE 64 "

you could retype the line. But try using the screen editor instead. First, type in the program and RUN it. Now LIST the program on the screen. Pressing SHIFT together with the CRSR key with the up and down arrows on it will make the flashing cursor move up one line. If you hold these two keys down the movement will be repeated until the keys are released. If you press the up/down CRSR key on its own without the SHIFT key then the flashing cursor on the screen will move down by one line. This function will also be repeated if you continue to hold the key.

Using these facilities you should move the cursor on the screen until it lies over the 3 of line 30 (the line that needs changing). Now using the left/right CRSR key move the flashing cursor along line 30 until it lies over the letter R in COMMORODE. You can then just type in the correct letters and they will automatically replace the old letters on the screen. Then press RETURN.

You can also insert extra spaces into a line in order to fit in missing letters. To open up an extra space in a line you should press the SHIFT key together with the key marked INST/DEL (INserT/DELete) at the top right of the keyboard. In this case, the letters RODE and the closing quote at the end of the line would move one place to the right. You can then enter an extra letter. In order to tell the Commodore that you have finished editing a line you must press RETURN with the flashing cursor standing somewhere on the edited line. If you do not press RETURN then the computer will not make any of your changes to the program:

PROGRAM DURING EDITING



You will frequently want to add lines to a program after you have written the first draft. Perhaps you forgot to put PRINT CHR$(147) at the beginning to start the program off on a clear screen. You do not have to edit any line numbers to do this. In the above program, for example, you can enter the new first line by typing:

5 PRINT CHR$(147)

As the computer executes BASIC instructions in line number order, this line will be carried out first.

First steps in bug-hunting

Mistakes in programs are called bugs, and the business of getting rid of them, debugging. As you have probably discovered, the Commodore helps a great deal in debugging programs by examining what you type in for errors in spelling and grammar or syntax. If it finds any, it alerts you by displaying an error message on the screen. You have already seen BAD SUBSCRIPT ERROR and SYNTAX ERROR on page 14. In fact the Commodore can display 28 different error messages. You may have come across some of them if you have made any mistakes in keying in any of the programs. In most cases the error messages will even tell you which lines of your program the errors are in. To correct an error, LIST the appropriate part of the program and edit it using the CRSR and INST/DEL keys.

Here are some programs which will not work. Try RUNning them and then checking the error messages they produce on the table that follows. You should then be able to find out which line is causing the problem in each of the programs:

BUGGED PROGRAMS

```
LIST

10 FOR X=60 TO 100
20 PRINT 2↑X,3↑X
30 NEXT X
READY.
```

```
LIST

10 FOR X=0 TO 20
20 INPUT B
30 PRINT B/X
40 NEXT X
READY.
```

BUGGED PROGRAM

```
LIST

10 INPUT "ENTER NUMBER ";A
20 FOR F=1 TO 12
30 PRINT TAB(9);F;"*";A;"=";F*A
40 NEXT G
READY.
```

COMMODORE ERROR MESSAGES

These are some error messages that you may encounter when writing your first programs.

Bad subscript
Specifically, this refers to an "array element" number out of range, but you may get this message with a simple typing mistake (see page 14).

Break
A program has been halted because you have pressed the RUN/STOP key. The message will show where the program was interrupted.

Device not present
This will appear if you attempt to record or play back a program without a tape recorder or disk drive connected (see page 58). It will also appear if the disk drive is not switched on!

Division by zero
You have asked the computer to perform a calculation which includes division by zero – a mathematical impossibility.

Formula too complex
This will appear if you write a calculation which has too many brackets for the computer to work with.

Illegal quantity
A number that you have used with a function is outside a permitted range – SQR with a minus number, for example.

NEXT without FOR
This will appear when a FOR ... NEXT program loop incorrectly uses two variables instead of one, or when FOR ... NEXT loops are incorrectly nested (see pages 28 and 56–57).

Overflow
A number produced by a calculation is too large (see page 19).

Redo from start
In a program using INPUT (see pages 26–27) you have keyed in a character when the computer was instructed to expect a number. The computer will now wait for a number.

String too long
A string that you have programmed the computer to produce exceeds its capacity of 255 characters.

?Syntax error
The computer cannot recognize what you have typed in.

Type mismatch
You have mixed up numbers and strings (see pages 14–15).

COMPUTER CONVERSATIONS

In all the programs you have written so far, you have given the computer a set of instructions and then left it to carry them out. Each program had just one outcome, which was exactly the same every time the program was RUN. But few real programs are like this; in a games program, for example, the player feeds the computer with new instructions every time the game RUNs. The computer takes in these instructions during the course of the game, changing the display in response to this input of information.

Indeed, it is difficult to write a program of any complexity without being able to interrupt the program while it is RUNning to feed in new information.

The BASIC word INPUT is intended to deal with this situation. It lets you carry on a conversation with the computer – you "talk" to it through the keyboard and it "talks" to you through the screen.

The INPUT command makes the computer remember information typed in on the keyboard, and gives it a name – a numeric variable if the information is a number, or a string variable if the information is made up of letters. Once the computer has labeled the information, it can then be passed on to later parts of a program. Here is an example of INPUT at work:

INPUT PROGRAM

```
LIST

10 PRINT CHR$(147)
20 PRINT "WHAT IS YOUR NAME"
30 INPUT NAME$
40 PRINT CHR$(147)
50 PRINT"**********************"
60 PRINT NAME$;"'S INPUT PROGRAM"
70 PRINT "**********************"
READY.
```

Questions from your computer

The program instructs the computer to display the question WHAT IS YOUR NAME. Line 30 then puts a question mark on the screen to indicate that the computer is waiting for new information from you. There's no need to hurry – there isn't a time limit. The computer will wait forever or until you type in the information it needs, whichever comes first. Type in your name and press the RETURN key.

The INPUT line of the program takes your name and labels it with the string variable NAME$. The dollar sign shows that the computer has been programmed to expect one or more letters.

This program may look familiar to you. It's similar to one that was featured on page 22 as an example of how to use LIST. If you compare it with that example, you will notice that INPUT can do the jobs done by two lines in the first INPUT program here. It can cause the question to be PRINTed and halt the program to await your response. A question mark is automatically PRINTed by INPUT – you don't have to type it in yourself. The next two screens show how INPUT works like PRINT. Note the semi-colon that appears before NAME$ in line 20:

USING **INPUT** TO **PRINT** A QUESTION

```
LIST

10 PRINT CHR$(147)
20 INPUT "WHAT IS YOUR NAME";NAME$
30 PRINT CHR$(147)
40 PRINT"**********************"
50 PRINT NAME$;"'S INPUT PROGRAM"
60 PRINT "**********************"
READY.
```

```
**********************
JAMES'S INPUT PROGRAM
**********************

READY.
```

Using INPUT to gather numbers

You can also use INPUT to gather numbers as a program is RUN. This has many practical applications.

Consider, for example, the problem of converting lengths, sizes or weights from one unit of measurement into another. The conversion is always the same – 2.54 centimeters to the inch, 0.3048 meters to the foot, 2.2 pounds to the kilogram and so on – but the numbers in each new calculation are different. Here is a simple conversion program for you to try out:

CONVERSION PROGRAM

```
LIST

10 PRINT CHR$(147)
20 PRINT " CONVERSION PROGRAM"
30 PRINT : INPUT "HOW MANY MILES";M
40 PRINT : PRINT M;" MILES = ";M*1.61;"
KILOMETERS"
READY.
```

The program asks you how many miles you want to convert to kilometers, waits for your reply, does the calculation and then displays the result on the screen. Because the INPUT line is expecting a number in response to the question it asks, the variable it produces is a numeric one.

Output formatting

You may notice that the output from the above program appears on a single line, with the various numbers and strings PRINTed fairly close together.

Now try editing line 40 above to change the semi-colons into commas:

MODIFIED CONVERSION DISPLAY

```
 CONVERSION PROGRAM
 HOW MANY MILES? 12

 12          MILES =    19.32       KILOMETER
S

READY.

LIST

10 PRINT CHR$(147)
20 PRINT " CONVERSION PROGRAM"
30 PRINT : INPUT "HOW MANY MILES";M
40 PRINT : PRINT M," MILES = ";M*1.61,"
KILOMETERS"
READY.
```

Now the output spreads out onto two lines. This is because the screen width is divided into four invisible zones or columns. Each of these zones is 10 characters wide. Using commas in the PRINT statement to separate items to be output causes each item to appear starting in a new zone. Keying in the examples on the next screen makes this clear:

SCREEN **PRINT** ZONES

```
PRINT 1,2,3,4,5,6,7,8
 1          2          3          4
 5          6          7          8

READY.

PRINT 1,2,3,4,5,6,7,8
 1          2          3          4
 5          6          7          8

READY.
```

More about TAB

Although the computer will automatically position numbers and strings either close-spaced or in zones, you are not limited to PRINTing them in this way. Indeed, as you saw on page 20, you may start PRINTing anywhere on a line using the BASIC function TAB. This function is always followed by a number in brackets which determines where an item will appear:

```
PRINT TAB(2);"TAB 2"
```

displays TAB 2 beginning 2 spaces in from the left. Here are some more examples of TAB in use:

USING **TAB**

```
PRINT TAB(5);"TAB (5)"
    TAB (5)
READY.
PRINT TAB(15);"TAB (15)"
              TAB (15)
READY.
PRINT TAB(25);"TAB (25)"
                        TAB (25)
READY.
PRINT TAB(30);"TAB (30)"
                             TAB (30)
READY.
```

WRITING PROGRAM LOOPS

Computers are extremely good at doing lots of simple, repetitive jobs very quickly. But if it is to do anything involving repetition, a computer must have some way of carrying out the same program or part of a program more than once. On page 23 you came across a loop using GOTO. Here it is in a slightly more complex loop (line 10 simply sets up the colors):

NEVER-ENDING LOOP PROGRAM

```
LIST

10 POKE 53280,6:POKE 53281,7
20 PRINT CHR$(144);CHR$(147)
30 LET X=1
40 PRINT X,X*2,X**X
50 LET X=X+1
60 GOTO 40
READY.
```

If you RUN this program, you will quickly see the disadvantage of using GOTO alone – the program is never-ending. Press STOP to stop it. The screen will show at which line number the program was stopped:

STOPPED LOOP

```
108         216         11664
109         218         11881
110         220         12100
111         222         12321
112         224         12544
113         226         12769
114         228         12996
115         230         13225
116         232         13456
117         234         13689
118         236         13924
119         238         14161
120         240         14400
121         242         14641
122         244         14884
123         246         15129
124         248         15376
125         250         15625
126         252         15876
127         254         16129
128         256         16384

BREAK IN 40
READY.
```

How to exit from a loop

The solution to these endless programs is the FOR ... NEXT loop. This allows you to set limits on how many times the loop is carried out. You can use it to PRINT the same table as the first loop program:

FOR ... NEXT LOOP PROGRAM

```
LIST

10 POKE 53280,6:POKE 53281,7
20 PRINT CHR$(144);CHR$(147)
30 FOR X=1 TO 20
40 PRINT X,X*2,X**X
50 NEXT X
READY.
```

The FOR ... NEXT loop both improves the program and shortens it by one line. Note that you don't have to set X equal to 1 or add 1 to it on each loop of the program now, because FOR ... NEXT takes care of this automatically. It starts off at line 30 by setting X equal to 1. Line 50 asks for the next value of X and so the program re-starts from line 30. This continues until X has a value of 20, the maximum set by line 30. In this case, the program stops, because its last line is line 50.

If necessary, the loop can be interrupted on each pass through to wait for new information. Try using INPUT in the middle of a FOR ... NEXT loop:

FOR ... NEXT/INPUT PROGRAM

```
LIST

10 FOR N=1 TO 5
20 PRINT CHR$(147)
30 PRINT : PRINT TAB(8);"TEMPERATURE CON
VERSION"
40 POKE 214,10 : PRINT : INPUT "GIVE ME
A FAHRENHEIT TEMPERATURE";TEMP
45 POKE 214,18 : PRINT : POKE 211,5
50 PRINT TEMP;"FAHRENHEIT = ";(TEMP-32)*
5/9;"CENTIGRADE"
60 FOR A=1 TO 5000
70 NEXT A
80 NEXT N
READY.
```

This program converts Fahrenheit temperatures into Centigrade. The FOR ... NEXT loop beginning at line 10 sets a limit of five calculations, after which you will have to RUN the program again. The INPUT statement at line 40 stops the program until you type in

the Fahrenheit temperature you want to convert. Line 50 then does the calculation and PRINTs the result.

How to slow your programs down

You might be confused by lines 60 and 70. They are to prevent the computer PRINTing and clearing results faster than you can read them. The two lines form a time delay to keep each result on the screen for a few seconds before continuing. This loop doesn't do anything other than divert the computer from the rest of the program. It is normally written as a single line:

```
60 FOR A=1 TO 5000: NEXT A
```

You can even miss out the final A, as the Commodore will assume that the NEXT refers to the FOR that precedes it. In general, a colon can be used in this way to separate commands on a single line instead of writing them on a number of different lines. Putting one loop inside another like this is called "nesting" loops. When you nest loops in your own programs, make sure there is a NEXT for every FOR.

How to round numbers off

When you RUN the temperature conversion program try entering a temperature of 32. The program tells you that 32 degrees Fahrenheit is equal to zero degrees Centigrade as you would expect. Next time the program asks for a temperature enter 34 degrees. The computer now tells you that 34 Fahrenheit is equal to 1.11111111 Centigrade, splitting the word Centigrade over two lines and spoiling the display. But there is a way to prevent this problem. It is sufficiently accurate for most purposes if the answer is given just as a whole number of degrees which then does not take up so much room on the line. Try using the screen editor to replace the expression (TEMP–32)*5/9 in line 50 by the slightly more complex expression INT((TEMP–32)*5/9). You'll find this will fix the problem:

ROUNDED DOWN CONVERSION DISPLAY

```
TEMPERATURE CONVERSION


GIVE ME A FAHRENHEIT TEMPERATURE? 34


34 FAHRENHEIT =  1 CENTIGRADE
```

The number is more sensible and the display looks much better. INT, short for INTeger, converts a decimal number into a whole number. If the result is, for example, 1.11111111, adding INT changes that to 1, an approximation that is quite accurate enough for most purposes. But when you use INT, remember that it always rounds numbers down. This can have a distorting effect with numbers that have large decimal fractions, as the next program shows:

USING INT

```
5 POKE 53280,2:POKE 53281,6
10 PRINT "INT(1.11)=";INT(1.11)
20 PRINT "INT(-3.1)=";INT(-3.1)
30 PRINT "INT(2.99)=";INT(2.99)


RUN
INT(1.11)= 1
INT(-3.1)=-4
INT(2.99)= 2

READY.
```

The first example is similar to the one you have already seen. The second example may seem a little surprising unless you remember that −4 is less than −3.1 and that INT rounds downward. The last example gives the expected answer, but looking back at the conversion program it would be better if 2.99 was rounded up to 3 not rounded down to 2. This can be achieved if you modify line 50 again:

COMPLETED CONVERSION PROGRAM

```
LIST
10 FOR N=1 TO 5
20 PRINT CHR$(147)
30 PRINT : PRINT TAB(8);"TEMPERATURE CON
VERSION"
40 POKE 214,10 : PRINT : INPUT "GIVE ME
A FAHRENHEIT TEMPERATURE";TEMP
45 POKE 214,18 : PRINT : POKE 211,5
50 PRINT TEMP;"FAHRENHEIT = ";INT((TEMP-
32)*5/9+0.5);"CENTIGRADE"
60 FOR A=1 TO 5000
70 NEXT A
80 NEXT N
READY.
```

By adding 0.5 to the temperature, you can ensure that INT produces the nearest whole number for each conversion.

DECISION-POINT PROGRAMMING

On the previous two pages you saw how loops can be used to make a program carry out the same sequence of commands a number of times. If you want to carry out a calculation or put something on the screen 10 times for example, you could write:

```
FOR A=1 TO 10 ...
NEXT A
```

But there is another way of doing this, by using an IF ... THEN statement. Let's say you want to PRINT all the numbers from 1 to 10, together with their squares and cubes in a table. First, here is how you would do it with FOR ... NEXT:

FOR ... NEXT LOOP

```
LIST

10 POKE 53280,2:POKE 53281,2
20 PRINT CHR$(147);CHR$(5)
30 PRINT " A"," A↑2"," A↑3"
40 PRINT "-----------------------------" :
 PRINT
50 FOR N=1 TO 10
60 PRINT N,N*N,N*N*N
70 NEXT N
READY.
```

```
A          A↑2        A↑3
-----------------------------

1          1          1
2          4          8
3          9          27
4          16         64
5          25         125
6          36         216
7          49         343
8          64         512
9          81         729
10         100        1000

READY.
```

In the IF ... THEN program which follows, line 10 sets up the color screen and line 30 PRINTs the table's heading as before. Line 60 is the first line of the loop – it increases N by 1 on every pass round the loop. Line 70 is the same PRINT statement used in the FOR ... NEXT program. Line 80 is where the computer makes a decision as it examines N. The < symbol is mathematical shorthand for "less than". So, if N is less than 10, the computer is told to go around the program again from line 60 (note that GOTO can be left out). When N is 10 the program ends:

IF ... THEN LOOP

```
LIST

10 POKE 53280,2:POKE 53281,2
20 PRINT CHR$(147);CHR$(5)
30 PRINT " A"," A↑2"," A↑3"
40 PRINT "-----------------------------" :
 PRINT
50 N=0
60 N=N+1
70 PRINT N,N*N,N*N*N
80 IF N<10 THEN 60
READY.
```

Why use the IF ... THEN loop?

You might wonder what the point of this is, as the IF ... THEN loop produces just the same results as the FOR ... NEXT loop. The value of IF ... THEN is that the computer can respond to any information that you INPUT during the program's operation by making a decision about it. Here is an example which shows this, by giving you a chance to test your skill at mental arithmetic (the new command RND in line 70 is explained on pages 52–53):

MATH TEST PROGRAM

```
LIST

10 PRINT CHR$(147);CHR$(144): POKE 53280
,5:POKE 53281,1
20 FOR Y=1 TO 12
30 FOR X=1 TO 40
40 PRINT "";
50 NEXT X : NEXT Y
60 POKE 214,8 : PRINT : POKE 211,13
70 X=INT(RND(0)*10) : Y=INT(RND(0)*10) :
 Z=INT(RND(0)*10)
80 POKE 214,15 : PRINT : POKE 211,11
90 PRINT "
100 POKE 214,15 : PRINT : POKE 211,11
110 PRINT X;"*";Y;"**";Z;"= ";: INPUT A
120 IF A=X*Y*Z THEN 160
130 POKE 214,15 : PRINT : POKE 211,11
140 PRINT "    WRONG
150 FOR T=1 TO 500 : NEXT T : GOTO 60
160 POKE 214,15 : PRINT : POKE 211,11
170 PRINT "    CORRECT
180 FOR T=1 TO 500 : NEXT T : GOTO 70
READY.
```

Each time the computer sets the problem and waits for your answer, it is faced with two possible courses of action. If you type in a correct answer, the IF ... THEN statement at line 120 directs the computer to go, not to line 130, but to line 160 next – PRINTing a "correct" message and then setting another problem. If the answer is wrong, then the computer "falls through" the IF ... THEN statement to line 130 and goes into the "wrong" routine.

It is important to remember that there must also be something in the program to stop the wrong answer routine carrying on into the correct answer routine. In this case, it is line 150, which makes the computer PRINT out the problem again after a delay:

MATH TEST DISPLAY

Selecting the right condition

When you use IF ... THEN, remember that there is a great variety of "conditions" which can follow the IF part of the statement. The programs on these pages have used either < or =, but this is only part of the complete range of symbols that the Commodore uses, as you can see from the table below.

You might think that you can only use IF ... THEN for comparing one number or numeric variable with another. However, this is not the case. The same set of conditions can be used with strings. A line like:

30 IF A$="FRED" THEN ...

at least makes some kind of sense. But what does the next line mean?

30 IF A$>"FRED" THEN ...

To answer this, you need to know how Commodore BASIC compares two strings.

Putting words in order

When the Commodore compares two strings, it takes the first character from each string and examines their ASCII codes. The string whose first character ASCII code is biggest is considered to be "larger" than the other. This means that FRED would be larger than ALAN because the ASCII code for F is greater than the code for A. If the first characters of both strings are the same, then the second letters are taken and compared, and so on, until a difference is found. With two words like ON and ONLY, where all the letters they share are the same, the longer word is considered to be greater than the shorter word. Only when two words are identical does the computer decide that they are equal.

If you look at the list of ASCII codes on page 61, you will see that the letters of the alphabet appear in order with codes from 65 to 90. This means that IF ... THEN can be used to sort words into alphabetic order, as this program shows:

ALPHABETIC SORTER

```
LIST

10 PRINT CHR$(147)
20 PRINT : INPUT "ENTER WORD 1";A$
30 INPUT "ENTER WORD 2";B$
40 IF A$=B$ THEN PRINT "THEY ARE THE SAM
E STRING" : GOTO 10
50 IF A$>B$ THEN PRINT A$;" COMES AFTER
";B$ : GOTO 20
60 PRINT A$;" COMES BEFORE ";B$ : GOTO 2
0
READY.
```

```
ENTER WORD 1? COMMODORE
ENTER WORD 2? BASIC
COMMODORE COMES AFTER BASIC

ENTER WORD 1? BASIC
ENTER WORD 2? COMMODORE
BASIC COMES BEFORE COMMODORE

ENTER WORD 1? MARCH 14
ENTER WORD 2? MARCH 3
MARCH 14 COMES BEFORE MARCH 3

ENTER WORD 1? MARCH 14
ENTER WORD 2? MARCH 03
MARCH 14 COMES AFTER MARCH 03

ENTER WORD 1? 14 OCTOBER
ENTER WORD 2? OCTOBER 14
14 OCTOBER COMES BEFORE OCTOBER 14

ENTER WORD 1?
```

IF ... THEN CONDITIONS

These symbols specify the kind of decision that the computer will make about what follows an IF command.

=	is equal to	<>	is not equal to
>	is greater than	<	is less than
>=	is greater than or equal to	<=	is less than or equal to

POKE AND PEEK

On some microcomputers a lot of memory space is given over to the BASIC interpreter so that the machine understands a large vocabulary of BASIC keywords. With the Commodore, the BASIC vocabulary is fairly small, so that it uses up a minimum amount of memory space, allowing room for larger programs. However, you still need to be able to program the computer to produce graphics and sound, for example. Both these facilities are controlled in other machines by keywords such as COLOR, DRAW and SOUND, none of which work on the Commodore. Instead, graphics, sound and a wide range of other functions are carried out by two general-purpose keywords – POKE and PEEK.

Using POKE to change colors

The POKE command is used to put a number directly into a location in the Commodore's RAM. If you type in the following program, you can see the effect of changing the values in just two of these memory "addresses":

COLOR CHANGE PROGRAM

```
LIST

5 PRINT CHR$(147)
10 FOR C=1 TO 15
20 POKE 53280,C : POKE 53281,C
30 FOR K=1 TO 1000
40 NEXT K
50 NEXT C
READY.
█
```

When you RUN this program, you should see that the screen border and background change through a sequence of colors. In fact the program cycles through all the colors that the Commodore can display. The actual color changes themselves are controlled by the two POKE statements in line 20. What this line does is to tell the computer to place or POKE the value of the variable C into the two memory locations 53280 and 53281. The Commodore looks at the contents of these two particular memory addresses to tell it which border color to put on the screen (53280) and which background color to use (53281). The 16 different colors that can be used are numbered from 0 to 15. The FOR … NEXT loop beginning at line 10 gives the variable C values from 1 to 15, so that it runs through all the colors except black, which is 0.

The codes for all of the colors are shown on the table at the bottom of this page. You should now be able to see how POKE can be used to turn the screen black, as mentioned on page 16. To restore the screen to its normal state after RUNning this program, press and hold the RUN/STOP key and at the same time press the RESTORE key (farthest right on the second row).

With so many separate memory addresses available on the Commodore, you cannot get very far by POKEing values at random. At worst, this can make the computer "hang", meaning that it goes into a state from which you have to switch off power to recover control. Of course, you will also lose whatever is in the computer's memory at the time. But there are many addresses that you can usefully POKE values into, and indeed all sound functions on the Commodore are controlled in this way, as you will see later on in this book.

Controlling key functions with POKE

To see another example of POKE at work, try typing the POKEs on the following screen, holding down a letter key after you press RETURN each time (don't worry about the error message):

SETTING AUTOREPEAT WITH POKE

```
POKE 650,128
READY.

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA

?SYNTAX  ERROR
READY.

POKE 650,0
READY.

A█
```

POKE COLOR CODES

All the Commodore's 16 colors can be produced by POKE commands ending with color control codes.

Color	POKE code	Color	POKE code
Black	0	Orange	8
White	1	Brown	9
Red	2	Light red	10
Cyan	3	Dark gray	11
Purple	4	Medium gray	12
Green	5	Light green	13
Blue	6	Light blue	14
Yellow	7	Light gray	15

As you may already have noticed, a few keys automatically repeat their function if they are held down. This is true for the space bar and the INST/DEL and CRSR keys. Key autorepeat, as this facility is known, can be very useful to have on the whole keyboard. Memory address 650 is the one that the Commodore uses to determine which keys the autorepeat will work on. If you POKE the value 128 into location 650 then all the keys will autorepeat. The previous screen shows what happens if you hold down the A key. A value of 0 POKEd into the same address will restore the keyboard to normal.

As a final example of POKE at work here is a program that will take an X and Y screen co-ordinate pair from you and then move the screen cursor to the position you have specified and PRINT a letter X:

CURSOR CONTROL PROGRAM

```
LIST

10 PRINT CHR$(147)
20 INPUT "(0-38)   X=";X
30 INPUT "(0-24)   Y=";Y
40 PRINT CHR$(147)
50 POKE 214,Y : PRINT : POKE 211,X
60 PRINT "X"
READY.
```

Here, address 214 holds the cursor Y position and address 211 holds the current cursor X position. You will find this facility for moving the cursor around the screen very useful when you program animation.

Looking into the memory with PEEK

PEEK is a keyword which has exactly the opposite effect of POKE. So, for example, if you type:

```
PRINT PEEK (650)
```

the number PRINTed will tell you whether key autorepeat is switched on or not. The value will be 128 if autorepeat is on for the whole keyboard, or 0 if it is just on for the editing keys and space bar.

PEEK is widely used in Commodore programming for taking the numbers from memory locations, modifying them in some way, and then POKEing them back again. You can use a loop with PEEK to see the values in the Commodore's memory. The next program does this. It's an endless loop which will go through all the memory locations – this screen shows the listing and just the first few PEEKs:

PEEK VALUES PROGRAM

```
10 POKE 53280,6:POKE 53281,3:PRINT CHR$(
144)
20 X=1
30 PRINT X;":";PEEK(X)
40 X=X+1:GOTO 30
READY.
RUN

 1 : 55
 2 : 0
 3 : 170
 4 : 177
 5 : 145
 6 : 179
 7 : 34
 8 : 34
 9 : 0
 10 : 0
 11 : 76
 12 : 0

BREAK IN 30
READY.
```

One very useful function that PEEK can carry out is examining the Commodore's real-time "jiffy" clock. The jiffy clock is a 3-byte binary counter. When the computer is switched on the jiffy clock is set to 0 and incremented by one, sixty times per second, for as long as the machine remains on.

The jiffy clock is stored at addresses 160, 161 and 162. Location 162 changes the fastest, at sixty times per second. Location 161 changes the next fastest, every time the value in location 162 is incremented from 255 back to 0 again. Finally, location 160 is incremented the slowest, every time the value in location 161 changes from 255 back to 0. Using just location 162 on its own will allow the Commodore to count just over 4 seconds before the clock starts again. Using locations 162 and 161 allows it a count of just over 18 minutes and with all three locations the machine can count to about three and a quarter days. You can see how PEEK can be used with the jiffy clock if you try out the next program, which times how long you take to answer a question:

JIFFY CLOCK PROGRAM

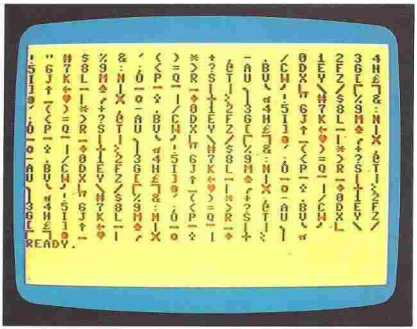
```
LIST

10 PRINT : PRINT "WHAT IS YOUR NAME";
20 POKE 162,0 : POKE 161,0
30 INPUT NAME$      PRINT
40 T=INT((PEEK(162)+PEEK(161)*256)/60)
50 PRINT "YOUR ANSWER TOOK";T;"SECONDS"
READY.
```

KEYBOARD GRAPHICS

On page 17 you saw how characters and controls on the Commodore's keyboard are represented by ASCII code numbers inside the computer. You can use this coding system to make the Commodore display its "character set", that is, all the characters arranged in order of coding. Here is a program that PRINTs the characters with codes from 33 to 126:

CHARACTER SET 1

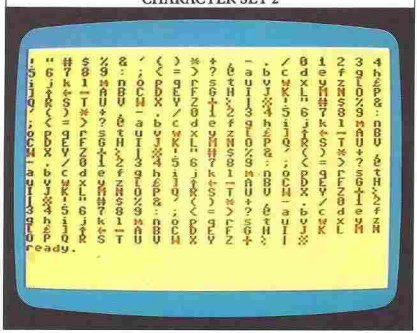
```
LIST

10 POKE 53280,6:POKE 53281,7:PRINT CHR$(
28);CHR$(147)
20 FOR N=1 TO 4
30 FOR X=33 TO 126
40 PRINT CHR$(X);" ";
50 NEXT X
60 NEXT N
READY.
```



This program actually repeats the set four times, filling the screen. You can see from the chart of characters on page 61 that this program PRINTs about half the set, up to code 126, and misses out the control characters with codes from 0 to 32.

The Commodore actually has two character sets. To see the second set you don't need another program. After RUNning the program above, just press the SHIFT and C= keys together. You should now see a character set which features capital letters with lower case letters as well:

CHARACTER SET 2



As you can see from these two displays, in addition to the letters A to Z and the numbers 0 to 9, the Commodore has very useful sets of predefined characters. You can key these in either directly, or in a program with PRINT, just as you would with letters.

If you look at the keyboard you will see that most of the keys have two of these graphics characters printed on the front of them. The right-hand one of the pair is accessed by holding the SHIFT key while the key with the required character on it is pressed. The left-hand character of the pair can be accessed in the same way but using the C= key instead of SHIFT. To get lower case letters, you should use the SHIFT and C= keys together to select the lower case character set.

How to reverse the graphics symbols

If you look at the keyboard, you will notice that the number keys 8 and 9 are marked RVS ON and RVS OFF. These keys enable you to produce the reverse of any character. To see this at work, press CTRL and 8, and then type in some characters on the screen. You will see that the text and screen colors are reversed, so that you get a character which is the inverse of the one normally produced. Pressing CTRL and 9 turns this off again. If you use CTRL and 8 within a string, it will appear as a reversed R control symbol. RVS ON and RVS OFF can also be activated with PRINT CHR$.

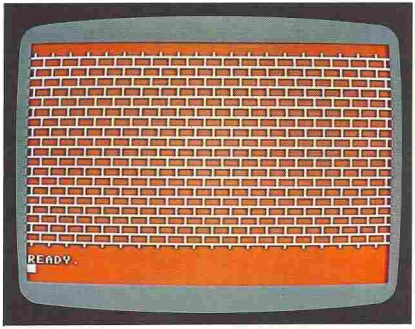
The keyboard graphics characters are arranged in groups with lines of varying thickness. If you cannot see the particular character you want to complete a design, this may be because what you actually need to key in is the reverse of another character. One reversed character that is particularly useful is the space. This produces a solid text square which can be built up into areas of color on the screen.

Designing with keyboard graphics

You can use the Commodore's graphics characters in programs to build up images on the screen. The following program uses only two of these characters, one shaped like a short capital letter T (line 50) and the other shaped like a capital T upside down (line 40). These two characters appear on the E and R keys:

KEYBOARD GRAPHICS PROGRAM

```
LIST
10 PRINT CHR$(147) : POKE 53280,12:POKE
53281,12
20 C=1 : PRINT CHR$(5);
30 FOR K=1 TO 20 : FOR D=1 TO 40
40 IF C=1 THEN PRINT"┴"; : GOTO 60
50 PRINT "┬";
60 C=1-C
70 NEXT D
80 C=1-C
90 NEXT K
READY.
```



If you RUN it, the program produces a reasonably good representation of a wall. If you change to the lower case character set again, the program will still RUN perfectly well. But if you type in programs with the lower case character set you must remember to type in all the BASIC keywords in lower case – without the SHIFT key pressed – otherwise the program will not work.

You can use the keyboard characters to build up other shapes on the screen by using loops. The next program, for example, produces a triangular stack of lines by PRINTing a series of characters. How many characters are PRINTed on each line is determined by the range of N:

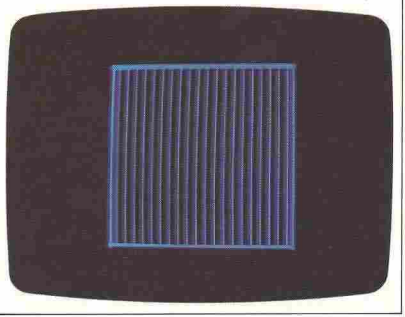
TRIANGLE PROGRAM

```
LIST
10 PRINT CHR$(147);CHR$(5)
20 POKE 53280,5:POKE 53281,0
30 FOR X=1 TO 20
40 FOR N=1 TO 2*X
50 PRINT "=";
60 NEXT N
70 PRINT
80 NEXT X
READY.
```

You can use loops to produce quite large shapes without having to write the entire shape out in the program. The next program draws and fills a rectangle:

RECTANGLE PROGRAM

```
LIST
10 PRINT CHR$(147);CHR$(31)
20 POKE 53280,0:POKE 53281,0
30 PRINT "                                        "
40 FOR Y=1 TO 18
50 PRINT "                                        "
60 NEXT Y
70 PRINT "                                        "
READY.
```



THE SCREEN MEMORY

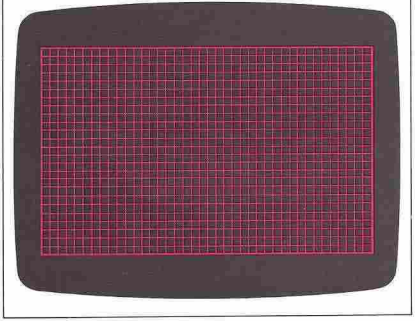
Now you know something about the Commodore's character sets, you can move on to a method for putting characters on the screen which is much more versatile than PRINT CHR$. This new technique uses POKE to fix both a character and a color for any position on the display.

Character and color maps

The Commodore's screen can be thought of as being divided up into a grid on which characters are displayed. Inside the computer there are two areas of memory set aside for the positions on this grid – the screen memory map and the color memory map. The first remembers the character to be displayed at each position on the grid, and the second remembers its color. Character and color can both be fixed by POKE commands.

Because the screen is divided into 25 lines of 40 characters, a total of 1000 positions, the character and color maps are each 1000 locations long. The character map begins at location 1024 and forms a continuous block up to location 2023. Similarly, the color map runs from location 55296 to 56295. The following grid shows how these positions are arranged (the grid on page 60 also gives the memory POKE numbers):

COMMODORE TEXT GRID



To put a character on the screen in a program, you need a line like this:

```
90 POKE 1464,81:POKE 55696,1
```

This puts a character in screen line 10 at position 0 across, that is at the left-hand edge. If you look up the character code, 81, in the table on page 60, you will see it is a solid circle or ball. The second POKE sets the color. Without this the ball would be invisible because it would be PRINTed in the same color as the screen. You can see from the color table on page 32 that 1, which appears at the end of this program line, is white.

An easy way to set memory locations

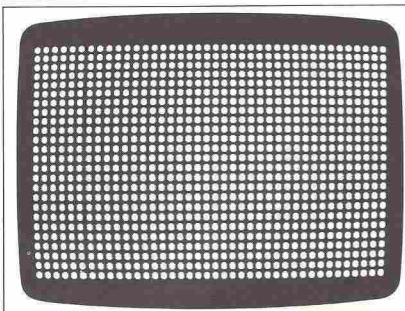
If you had to remember all these POKE numbers to use them each time in a program it would be a very lengthy process. However, knowing where the first locations in the two memory maps are, you can quickly work out the POKE number for any other point. If Y is the number of screen lines down, and X is the number of positions across, the following program line will produce a white ball at that position:

```
40 POKE 1024+Y*40+X,81:POKE 55296+Y*40+X,1
```

This will take any values of X (positions across) and Y (lines down) and produce the character, as long as the values are within the screen limits. By looping both X and Y you can produce characters all over the screen:

CHARACTER **POKE** PROGRAM

```
LIST

10 POKE 53280,11:POKE 53281,0
20 FOR Y=0 TO 24
30 FOR X=0 TO 39
40 POKE 1024+Y*40+X,81:POKE 55296+Y*40+X
,1
50 NEXT X
60 NEXT Y
READY.
```

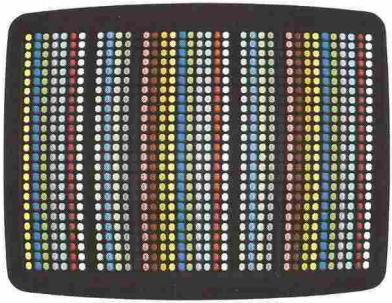


Now you can experiment by adding color. You could simply change from white to another color, but instead try altering the final character in line 40 so that it now reads:

```
40 POKE 1024+Y*40+X,81:POKE 55296+Y*40+X,X
```

Now the color depends on the position, so that it changes with every character. When X goes above 15, the computer starts the color series from the beginning again. The result is the same display as in the previous screen but this time with all the Commodore's colors:

COLOR CHARACTER **POKE**



Using POKE for graphics

By POKEing a character and color onto the screen, you can build up your own graphic displays. The following program allows you to draw designs on the screen. It's a simple sketch-pad listing that uses one of the text characters – an asterisk – to produce outline drawings. The program uses POKE to change a cursor position each time you key in an instruction:

POKE GRAPHICS PROGRAM

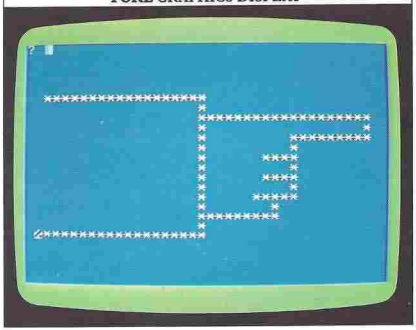
```
LIST -140

10 POKE 53280,13
20 POKE 53281,6
30 PRINT CHR$(147);CHR$(5)
40 X=19 : Y=12
50 C=32 : POKE 650,128
60 POKE 1024+Y*40+X,165
70 POKE 55296+Y*40+X,1
80 PRINT CHR$(19);" ";
90 PRINT CHR$(19); : INPUT A$
100 IF A$="L" THEN 200
110 IF A$="R" THEN 250
120 IF A$="U" THEN 300
130 IF A$="D" THEN 350
140 IF A$="B" THEN C=32 : GOTO 160

READY.
```

POKE GRAPHICS PROGRAM

```
LIST 150-

150 C=42
160 DX=0 : DY=0 : GOTO 400
200 IF X<2 THEN 80
210 DX=-1 : DY=0 : GOTO 400
250 IF X>38 THEN 80
260 DX=1 : DY=0 : GOTO 400
300 IF Y<2 THEN 80
310 DX=0 : DY=-1 : GOTO 400
350 IF Y>23 THEN 80
360 DX=0 : DY=1
400 POKE 1024+Y*40+X,C
420 X=X+DX : Y=Y+DY
430 GOTO 60
READY.
```

You can control the movement of the cursor with the U, D, L and R keys (for up, down, left and right respectively). Once you have selected the direction in which you want to move the cursor you should press RETURN. To take another step in the same direction, you need only press RETURN again. If you want to move the cursor several steps in any direction, the program is arranged so that it will autorepeat. This means that you only have to press and hold the RETURN key and the cursor will move rapidly in the last direction you have selected. To draw lines while moving the cursor enter an asterisk and press RETURN and then move the cursor in the normal way. To blank lines out again if you make a mistake enter a B and press RETURN and the cursor will then leave spaces behind it as it moves. The display is limited to the 40×25 character positions on the screen. Although the designs it can show are therefore at low resolution, it works quite quickly. Here is an example of the sort of display you can produce:

POKE GRAPHICS DISPLAY



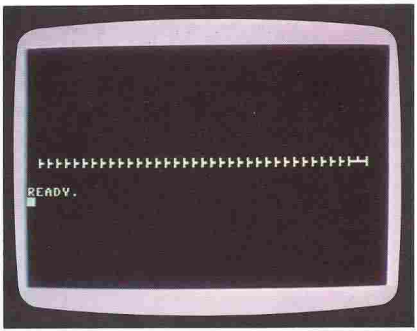
ANIMATION

Animation consists of programming the computer to place a character on the screen and then move it in a series of small steps. There are two ways of doing this, using either POKE with PRINT, or POKE for position, character and color. Both methods use loops to change the position of the character in ways which you can specify.

Here's a simple animation program which shows you some of the problems that have to be overcome when controlling movement:

ANIMATION WITH CHANGING CO-ORDINATES

```
LIST

10 PRINT CHR$(147);CHR$(5);
20 POKE 53280,4: POKE 53281,0
30 V=10 : FOR X=1 TO 35
40 POKE 214,V : PRINT : POKE 211,X
50 PRINT CHR$(171);CHR$(177),CHR$(179);
60 NEXT X : PRINT : PRINT
READY.
```



How to remove after-images and control speed

The first thing you will notice when you RUN this program is that it doesn't do exactly what you want it to. It moves the character from left to right by changing the position of the cursor with POKE but it doesn't remove the old image first, because you haven't told the computer to do that. This means that on every move, you get left with a small part of the previous image which isn't deleted. In some programs you may want to retain these after-images for special effects but more often you will want to remove them. Another thing you may have noticed is the speed with which the object moves across the screen – it is far too fast for most purposes. Both of these problems can easily be cured by keying in the following lines with the first program still in memory:

CHANGES TO IMPROVE DISPLAY

```
LIST

50 PRINT " ";CHR$(171);CHR$(177);CHR$(17
9);
55 FOR T=1 TO 10 : NEXT T
READY.
```

The space in line 50 erases the left-hand part of the image, while a delay loop slows the movement down.

Vertical movement can be produced by the same method. This time the X co-ordinate is kept constant, while the Y co-ordinate is changed by a loop. You can make an object move upward with a line like this:

```
50 POKE 214,24-Y:POKE 211,X
```

As Y increases, the object is PRINTed higher and higher up the screen. To erase after-images, spaces must be PRINTed by moving the cursor back one space each move.

How to program two-way motion

So far you have seen animation that uses FOR ... NEXT to vary either the X or Y co-ordinates of an object, moving it in just one direction. This idea works well as far as it goes, but it is not much use where, for example, you want to bounce a ball from side to side across the screen. The easiest and probably one of the best ways of achieving this two-way motion is to have the program set up an extra variable in addition to the X and Y co-ordinate variables. These new variables hold the direction of movement of the object. The following program uses this technique with the variable DX to move a ball from side to side across the screen:

BOUNCING BALL PROGRAM

```
LIST

10 PRINT CHR$(147);CHR$(5)
20 V=10 : X=20 : DX=1 : POKE 53280,6 : P
OKE 53281,12
30 POKE 214,V : PRINT : POKE 211,X
40 PRINT" "CHR$(113);" ";
50 X=X+DX
60 FOR T=1 TO 10 : NEXT T
70 IF X<1 THEN DX=-DX
80 IF X>36 THEN DX=-DX
90 GOTO 30
READY.
```

In this program, line 20 sets up the initial X and Y co-ordinates and also sets up the direction variable, DX, to start the movement off from left to right. Line 30 positions the cursor at the current X and Y co-ordinates on the screen. Line 40 PRINTs the ball and also erases any previously drawn ball characters to the left or right of the current position by PRINTing spaces there. Line 50 updates the X position by adding DX to it.

Because DX is currently 1 this update increases X by 1 and thus moves the ball one place to the right. If DX had been −1 then this same line would have moved the ball to the left by one place. Lines 70 and 80 reverse the ball's direction if it is at the screen edge by changing the sign of DX from +1 to −1 or vice versa.

Animation with POKE

As well as using PRINT CHR$ to produce animation, you can also POKE characters from the graphics character set straight onto the screen, erasing them again to produce movement:

COLOR TRACK PROGRAM

```
LIST -140

10 PRINT CHR$(147)
20 POKE 53280,6 : POKE 53281,6
30 FOR X=10 TO 30
40 POKE 1024+6*40+X,81 : POKE 55296+6*40
+X,6
50 POKE 1024+15*40+X,81 : POKE 55296+15*
40+X,6
60 NEXT X
70 FOR Y=7 TO 14
80 POKE 1024+Y*40+10,81 : POKE 55296+Y*4
0+10,6
90 POKE 1024+Y*40+30,81 : POKE 55296+Y*4
0+30,6 : NEXT Y
100 FOR X=10 TO 30
110 POKE 55296+6*40+X,1
120 FOR T=1 TO 5 : NEXT T
130 POKE 55296+6*40+X,6
140 NEXT X

READY.
```

COLOR TRACK PROGRAM

```
LIST 150-

150 FOR Y=7 TO 14
160 POKE 55296+Y*40+30,1
170 FOR T=1 TO 5 : NEXT T
180 POKE 55296+Y*40+30,6
190 NEXT Y
200 FOR X=10 TO 30
210 POKE 55296+15*40+(40-X),1
220 FOR T=1 TO 5 : NEXT T
230 POKE 55296+15*40+(40-X),6
240 NEXT X
250 FOR Y=7 TO 14
260 POKE 55296+(21-Y)*40+10,1
270 FOR T=1 TO 5 : NEXT T
280 POKE 55296+(21-Y)*40+10,6
290 NEXT Y
300 GOTO 100
READY.
```

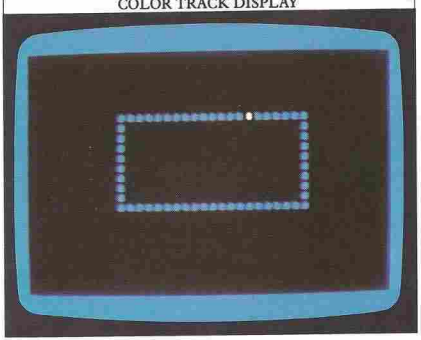
In this program, a ball speeds around a rectangular track. The listing looks quite involved, but when fairly complex motion is required, POKE is actually an easier programming technique than PRINT CHR$, even though it means having to program color locations as well as character numbers. It is also easier to develop programs by changing POKE values.

Lines 10 and 20 clear the screen and set the screen color to blue. Lines 30 to 90 now POKE a rectangular pattern of ball characters onto the screen again in dark blue. Lines 100 to 290 then change the color of each ball in turn, in a clockwise direction, from the dark blue background color to white, then, after a short delay, back to dark blue. If you type in the following line to change the screen color, this whole process then becomes visible:

```
20 POKE 53280,6:POKE 53281,0
```

When the change has been made, the display looks like this, with the white ball traveling around the blue track:

COLOR TRACK DISPLAY



USING A DATA BANK 1

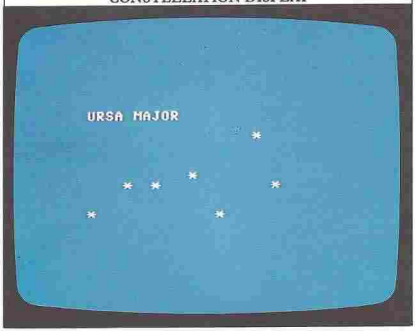
The data necessary for a program can be collected while it is RUNning by using INPUT, or alternatively can be written into the program itself. The commands used to store data are quite straightforward. Data is held in DATA statements and read by READ statements. This program shows you the technique at work:

CONSTELLATION PROGRAM

```
LIST
10 POKE 53280,6:POKE 53281,6:PRINT CHR$(
147);CHR$(5)
50 POKE 214,6 : PRINT : POKE 211,7
30 PRINT "URSA MAJOR"
40 DATA 8,25,12,18,13,11,13,14,13,27,16,
7,15,21
50 NUMBER=7
60 FOR N=1 TO NUMBER
70 READ Y,X
80 POKE 214,Y : PRINT : POKE 211,X
90 PRINT "*"
100 FOR T=1 TO 500 : NEXT T
110 NEXT N
120 PRINT CHR$(31)
READY.
```

When you RUN this program you should see on your screen a computer-generated map of a group of stars, the constellation Ursa Major, also known as the Big Dipper:

CONSTELLATION DISPLAY

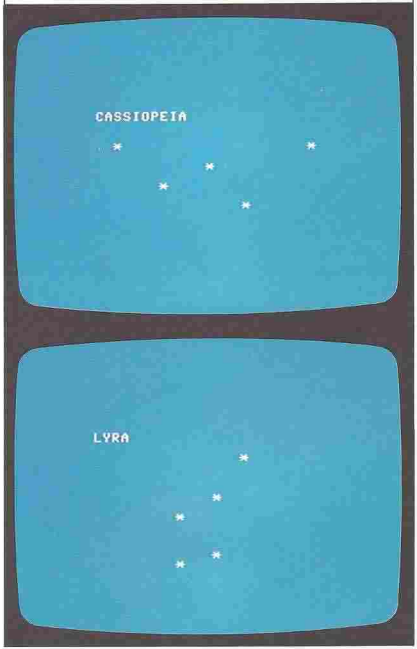


The information for the display is carried in line 40 in the form of 14 co-ordinates. Line 70 tells the computer to READ the DATA in line 40, and to understand the DATA as pairs of figures which the program will refer to as Y,X. Line 50 tells the computer that there will be seven of these pairs altogether.

With a program like this it is easy to enter new DATA to get the computer to PRINT a new map. Here are two sets of line changes and the maps they produce:

CONSTELLATION LINE CHANGES

```
LIST
30 PRINT "CASSIOPEIA"
40 DATA 9,9,9,30,11,19,13,14,15,23
50 NUMBER=5
READY.


LIST
30 PRINT "LYRA"
40 DATA 8,23,12,20,14,16,18,20,19,16
50 NUMBER=5
READY.
```



When you use DATA statements, it is important to tell the computer how much DATA there is to READ. Line 60 in the constellation program shows you how to do this. It sets the limit for the number of pairs of co-ordinates that are stored in the DATA line, so when the computer has PRINTed the final star, it stops. If there was no FOR ... NEXT loop controlling the READ command, the computer would run out of DATA. If this happened the program would end with an error message:

```
? OUT OF DATA ERROR IN 70
```

Storing numbers and words together

Words, too, can be stored and read using DATA lines, and you can also store a mixture of both numbers and words – the names of friends and their phone numbers or birthdays, for example. This does present a problem though, because two different types of DATA are to be used, numbers and strings. But careful organization of the DATA and READ statements can get round this:

TELEPHONE LIST PROGRAM

```
LIST -180

10 PRINT CHR$(147)
20 DATA J.BAKER,322,G.HILTON,166,R.HERMA
NN,103,S.KLEIN,191
30 DATA M.PERERA,86,T.PHILLIPS,71,P.RICH
ARD,100
40 DATA D.SMITH,27
50 PRINT "PERSONAL TELEPHONE LISTING"
60 FOR T=1 TO 2000 : NEXT
70 PRINT CHR$(147) : POKE 214,4 : PRINT
80 PRINT "COMPLETE LIST.....PRESS 1"
90 PRINT "SELECTIVE LIST....PRESS 2
100 INPUT "CHOICE";CHOICE
110 IF CHOICE=2 THEN 190
120 PRINT CHR$(147)
130 PRINT : PRINT : PRINT
140 FOR C=1 TO 8
150 READ NAME$,N
160 PRINT TAB(6);NAME$;TAB(20);N :PRINT
170 NEXT C
180 END

READY.
```

```
LIST 190-

190 PRINT CHR$(147) : POKE 214,4 : PRINT
    : PRINT "ENTER INITIAL AND NAME"
200 INPUT ENTRY$
210 RESULT$="NAME NOT FOUND"
220 FOR C=1 TO 8
230 READ NAME$,N$
240 IF NAME$=ENTRY$ THEN RESULT$=NAME$+"
    "+N$
250 NEXT C : PRINT CHR$(147)
260 PRINT RESULT$ : FOR T=1 TO 5000
270 NEXT : RESTORE : PRINT CHR$(147): GO
TO 50
READY.
```

This program holds a personal telephone list. Names and telephone numbers are held in lines 20 to 40. Lines 50 to 90 display the program title and then offer a choice of functions. If you type:

```
1 RETURN
```

the computer PRINTs the entire telephone list:

TELEPHONE LIST DISPLAY

```
PERSONAL TELEPHONE LISTING
COMPLETE LIST.....PRESS 1
SELECTIVE LIST....PRESS 2
CHOICE? 1

J.BAKER          322
G.HILTON         166
R.HERMANN        103
S.KLEIN          191
M.PERERA          86
T.PHILLIPS        71
P.RICHARD        100
D.SMITH           27

READY.
```

If you type in:

```
2 RETURN
```

the program follows lines 190 to 270. You are first asked to enter an initial and a name.

If the computer finds that the name (ENTRY$) that you type in is the same as one of the names (NAME$) in the DATA statements, it will give a new string (RESULT$) the value of NAME$ plus a line of dots and the telephone number. If it does not find ENTRY$, RESULT$ is left unchanged at "Name not found" (set by line 210), and that is PRINTed out at the end.

Because you want to add the name, a line of dots and the telephone number together in line 240, the telephone number has to be treated as a string variable (N$) instead of the numeric variable (N) used in line 150. If you were to substitute N for N$ in lines 230 and 240, you would get back an error message:

```
? TYPE MISMATCH ERROR IN 240
```

because string and numeric variables cannot be added.

You can extend this program to hold a much longer list of your own names and numbers by putting them into the DATA lines, and then altering the limits of the two loops at lines 140 and 220. To see the complete list if it is more than one screen long, press CTRL as the program display begins to slow down the scrolling.

Line 270 uses a command which will be new to you – RESTORE. Without this, the program will RUN correctly only once – after this you will get an error message. You can find out why this happens on page 43.

USING A DATA BANK 2

On the Commodore, READing DATA can produce quite complex graphics when the DATA refers to keyboard graphics characters. On these two pages, you will see how you can store this sort of DATA most easily to produce static and animated displays.

The following program uses DATA to store the details of a maze. It also features a new way of using GOTO:

MAZE PROGRAM

```
LIST -170

10 POKE 53280,8:POKE 53281,0:PRINT CHR$(
147);CHR$(5):PRINT:PRINT
20 FOR Y=1 TO 13:FOR X=1 TO 13
30 READ M : PRINT TAB(12);
40 ON M GOTO 60,70,80,90,100,110
50 ON M-6 GOTO 120,130,140,150,160
60 PRINT CHR$(93); : GOTO 170
70 PRINT CHR$(99); : GOTO 170
80 PRINT CHR$(173); : GOTO 170
90 PRINT CHR$(176); : GOTO 170
100 PRINT CHR$(174); : GOTO 170
110 PRINT CHR$(189); : GOTO 170
120 PRINT CHR$(177); : GOTO 170
130 PRINT CHR$(171); : GOTO 170
140 PRINT CHR$(178); : GOTO 170
150 PRINT CHR$(179); : GOTO 170
160 PRINT " ";
170 NEXT X

READY.
```

```
LIST 180-

180 PRINT
190 NEXT Y
200 DATA 4,2,2,2,6,11,8,2,2,2,2,5
210 DATA 1,11,11,11,11,11,11,1,11,11,11,1,
11,1
220 DATA 1,11,4,2,5,11,3,2,2,2,5,11,1
230 DATA 1,11,1,11,1,11,11,11,11,11,11,1,11
,1
240 DATA 1,11,1,11,3,2,2,2,5,11,1,11,1
250 DATA 1,11,1,11,11,11,11,11,1,11,1,11,1
,1
260 DATA 1,11,3,2,11,2,5,11,1,11,1,11,11,1
270 DATA 1,11,11,11,11,11,1,11,1,11,11,1,11
,1
280 DATA 8,2,9,2,2,2,6,11,3,2,6,11,1
290 DATA 1,11,1,11,11,11,11,11,11,11,11,11,
11,1
300 DATA 1,11,3,2,11,2,2,2,2,5,11,1
310 DATA 1,11,11,11,11,11,11,11,11,11,11,1,
11,1
320 DATA 3,2,2,2,2,2,2,2,2,2,10,11,1
READY.
```

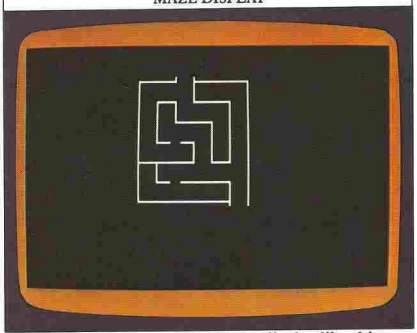
To draw the maze the program uses some of the permanently programmed keyboard graphics characters. Rather than use the ASCII codes for these characters in the DATA statements, the program gives each of the characters in the display a code of its own in the range 1 to 11.

When a lot of DATA has to be stored, reducing the size of the DATA block like this can be very valuable. The only problem with this is that the DATA needs to be decoded before it can be used. In this case the program needs to tell the computer which character to PRINT for each of the codes. This is dealt with in lines 40 and 50 using the pair of commands, ON ... GOTO.

The variable M in line 40 is the one that contains the coded DATA just READ in line 30. Line 40 says that if the value of M is 1 then GOTO the first line number in the list of numbers following the GOTO. If the value of M is 2 then the computer will GOTO the second line number in the list and so on. If M has a value greater than the number of lines in the line number list then line 40 is ignored and the computer moves on to line 50. This line subtracts 6 from M and then selects one of a new group of line numbers.

Each of the lines specified in the ON ... GOTO statements PRINTs the correct graphics character, and then directs the computer to READ the next coded character for the maze:

MAZE DISPLAY



Once you know how to produce a display like this, you can develop it for your own use. You can build up quite complex shapes with the graphics characters including charts and tables (looping the PRINT lines helps to save space when you do this).

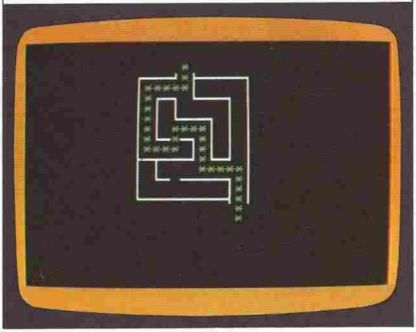
The Commodore's cursor keys will let you move the cursor through the maze to find the way out. By using a series of different mazes in conjunction with a timing routine, you can develop this system of DATA storage to create a simple game:

MAZE PROGRAM CODES

In the maze program, graphics are selected by short program character codes.

Character	Code	Character	Code
▯	1	▯	7
▯	2	▯	8
▯	3	▯	9
▯	4	▯	10
▯	5		
▯	6		

MAZE GAME DISPLAY



Using DATA for animation

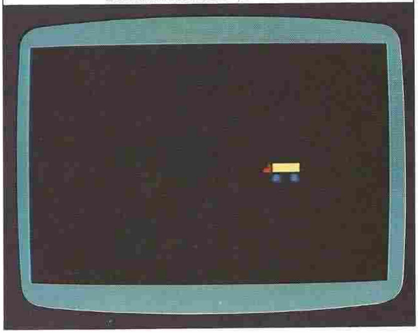
Another graphics area in which READing DATA can be helpful is in storing a graphics shape which can then be used for animation. In order to be able to continually move the same DATA over and over again, you need to make use of the RESTORE statement. Here is a program which features this technique:

DATA ANIMATION PROGRAM

```
LIST

10 PRINT CHR$(147)
20 POKE 53280,12:POKE 53281,0
30 X=10 : Y=7 : DX=1 : DY=-1
40 IF X<3 THEN DX=-DX
50 IF X>35 THEN DX=-DX
60 IF Y<2 THEN DY=-DY
70 IF Y>19 THEN DY=-DY
80 X=X+DX : Y=Y+DY : RESTORE
90 POKE 214,Y : PRINT : POKE 211,X
100 READ A$ : PRINT A$
110 POKE 214,Y+1 : PRINT : POKE 211,X+1
120 READ B$ : PRINT B$
130 FOR T=1 TO 10 : NEXT T
140 PRINT CHR$(147) : GOTO 40
150 DATA "[illegible]","[illegible]"
READY.
█
```

The program uses an animation method similar to one on page 39, but here the program is arranged in a different order. Line 10 clears the screen and line 20 sets up the colors. Line 30 sets up the initial values of the X and Y co-ordinates and the initial X and Y directions. Lines 40 to 70 perform the check for a co-ordinate out of range. Line 80 updates the X and Y co-ordinates with DX and DY and RESTOREs the DATA pointer (you'll see what this means in a moment). Lines 100 and 120 READ the graphics DATA out of the DATA statement in line 150 and PRINT and color the graphic shape onto the screen. Line 130 gives a short delay before line 140 erases the graphic shape by clearing the screen:

DATA ANIMATION DISPLAY



How to reset the DATA pointer

You can have any number of DATA statements anywhere in a listing. The Commodore treats them as though they are all joined together. Each time the computer comes across a READ command, it READs the next item of DATA in line. But what happens when the computer gets to the end of the DATA?

In programs where the DATA is only used once, that's no problem. But if you want the computer to use the same DATA a number of times, you need to use RESTORE to tell the DATA pointer to point back to the beginning of the DATA again. That's why RESTORE was needed in the telephone list program on page 41, and in the previous program on this page. Here's another program which shows you how just a small amount of DATA can be used repeatedly to work out tax totals:

TAX RATES PROGRAM

```
LIST

10 PRINT CHR$(147)
20 PRINT : INPUT "A=";A
30 PRINT
40 FOR C=1 TO 6
50 READ T
60 PRINT A;"+";T"% =";A+A*T/100
70 NEXT C
80 RESTORE
90 GOTO 20
100 DATA 10,12.5,15,17.5,20,25
READY.
█
```

INTRODUCING SPRITES

As you will have discovered by now, producing text graphics using PRINT and POKE has its drawbacks – chiefly that the screen resolution is rather limited. But the Commodore does have other methods of producing displays, and one of these, programming sprites, can give you much more detailed graphics for animation.

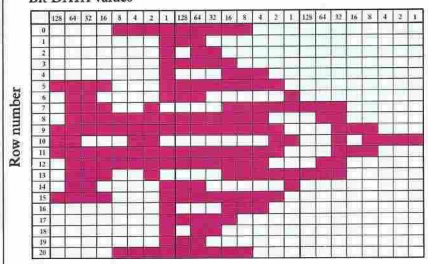
Sprites, or MOBs (Mobile Object Blocks), are each about the size of 9 ordinary text characters put together in a 3×3 block. You can program up to 8 different sprites to appear on the screen at once. The shape, position and color of the sprites is controlled by a special chip in the Commodore called the Video Interface Circuit, or VIC chip for short. To produce a sprite, all its details have to be POKEd into the relevant locations in this chip.

Designing a sprite

Each sprite is made up of 504 tiny picture elements ("pixels" for short). These pixels are arranged in a rectangular pattern consisting of 21 rows with 24 pixels on each row. To make a sprite, you first need to sketch the design on a grid with one square for each pixel (you will find a blank grid for this on page 59). The design can be any shape you like as long as it fits inside the grid. One possible sprite layout – an airplane – has been filled in here ready for programming:

USING A SPRITE GRID

The sprite design is laid out on a sprite grid so that DATA values can be calculated.



Having designed your sprite you now need to transfer this information into the computer in a form that it can understand. Everything inside the Commodore is arranged in bytes, each made up of 8 binary digits or bits. Each pixel in a sprite is controlled by one bit, so it takes 24÷8=3 bytes to control each complete row of the sprite, dictating which pixels are turned on or off. As there are 21 rows in a sprite, it takes 3×21=63 bytes to completely specify the pixels.

This information has to be POKEd into consecutive memory locations inside the computer where the VIC chip can "see" it and get at it. The easiest way to store this information is to put it in a series of DATA statements, and then make the program READ it and POKE it into the memory. In the following program, the DATA contains the pixel information for the airplane design. As yet, it does not contain any instructions for putting the sprite on the screen:

SINGLE SPRITE PROGRAM

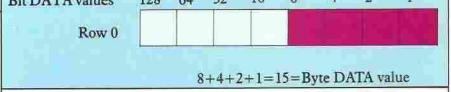
```
LIST

10 POKE 53280,6:POKE 53281,2
20 PRINT CHR$(147)
30 FOR C=0 TO 63
40 READ BYTE
50 POKE 832+C,BYTE
60 NEXT C
500 DATA 15,248,0,1,192,0,1,96,0
510 DATA 1,240,0,1,28,0,241,254,0
520 DATA 97,193,0,114,31,254,161,254,252,96
530 DATA 255,254,56,198,3,47,255,254,56
540 DATA 127,252,38,114,31,254,97,193,0
550 DATA 241,254,0,1,28,0,1,240,0
560 DATA 1,96,0,1,192,0,15,248,0,0
READY.
```

Entering sprite DATA

The first thing to notice about the program above is the technique used to arrive at the list of numbers in the DATA statements. This is where the numbers and titles in the blank spaces around the grid come in. Starting at the top left corner of the sprite design and working left to right across the top row, read the first 8 pixels and the numbers above them. In this row, the pixels in the right-hand half of the row are turned on, making up the wingtip, while those in the other half are turned off:

ADDING UP DATA BITS

Each 8-row unit of sprite pixels is entered in a program as a single number – the total of the individual pixel bit DATA values.



Each 0 indicates a pixel that is not lit, while a 1 indicates one that is lit. The 8 column numbers then have to be added together to form a byte. Here the pixels are lit in the columns headed 8, 4, 2 and 1, so the byte total is 15. This is the first number in the DATA statements. You

then move on to the next 8 pixels to the right, gradually working down through the grid. This gives a total of 63 DATA numbers. But because 63 is not a very convenient total for the computer to work with, a 64th byte has to be added. This is used just as "padding" – it does not actually specify anything, and is set to zero.

Placing sprites in the memory

The next thing the program does is to POKE the bytes into memory in line 50. In the standard Commodore memory map there is nowhere that is completely safe for sprite DATA storage. This program uses an area of the memory that is normally reserved for cassette input and output operations. It starts at location &28 but this is not a multiple of 64 so the program starts a bit higher at 832 (64×13). In fact there is room for up to 3 sprites in this area starting at memory locations 832, 896 and 960. The FOR ... NEXT loop between lines 30 and 60 READs each item of DATA, POKEing each into one of the memory locations after 832. When you RUN the program now, you should see the sprite appear :

SPRITE LOCATION LINES

```
LIST 70-110

 70 POKE 2040,13
 80 POKE 53287,1
 90 POKE 53248,50:POKE 53249,100:POKE 532
64,0
100 POKE 53269,1
110 POKE 53271,1:POKE 53277,1

READY.
```

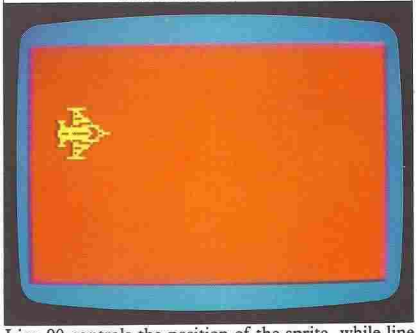
Within the Commodore's memory there are 8 locations which are used as signposts to tell the computer where the DATA for sprites has been stored. There is one byte for each sprite. As a single byte can hold any number from 0 to 255, this means that there are 256 possible locations that sprite DATA can start at. The starting location for the DATA for this sprite is specified by line 70. It directs the computer to the 13th block of 64 bytes, in other words the memory location 832.

You can see how color is controlled if you change line 80 to:

```
80 POKE 53287,7
```

If you then RUN the program, you should see the change in the display:

COLOR SPRITE DISPLAY



Line 90 controls the position of the sprite, while line 100 POKEs a number into the location which turns sprites on and off. Line 110 activates two memory locations which control the size of sprites. The two POKEs in this line make the sprite take up four times the amount of space on the screen that it would otherwise occupy.

You will find out more about how to use these different control locations on the next two pages. You will also find out how to make this program simpler – so before turning over, either SAVE your program on tape or disk (page 58 will tell you how to do this), or make sure that it's held in memory.

Why sprites don't work like text

Because sprites are under the control of the VIC chip, they don't behave in the same way as the text characters. You will find that if you RUN the program on this page, you cannot erase the sprite with SHIFT and CLR, and even if you LIST the program, the sprite will stay in the same position on the screen instead of scrolling upward. To get rid of the sprite you need to use the RUN/STOP and RESTORE keys.

PROGRAMMING WITH SPRITES

When you are programming sprites, you can find yourself writing lines that use a number of rather unmemorable POKE addresses for the VIC chip. But because the addresses in this chip run in a sequence from a lowest value of 53248 upward, you can avoid this problem by giving a variable the value 53248, and then using the variable instead of the number. You could do this with the program from the previous two pages by first putting in this line:

```
5 LET V=53248
```

This enables you to refer to any locations in the VIC chip in terms of V plus a one- or two-digit number. This makes sprite programs a lot easier to interpret. With this method, the sprite program from the previous pages now looks like this:

SIMPLIFIED SINGLE SPRITE PROGRAM

```
LIST

5 LET V=53248
10 POKE 53280,6:POKE 53281,2
20 PRINT CHR$(147)
30 FOR C=0 TO 63
40 READ BYTE
50 POKE 832+C,BYTE
60 NEXT C
70 POKE 2040,13
80 POKE V+39,1
90 POKE V,50:POKE V+1,100:POKE V+16,0
100 POKE V+21,1
110 POKE V+23,1:POKE V+29,1
500 DATA 15,248,0,1,192,0,1,96,0
510 DATA 1,240,0,1,48,0,241,254,0
520 DATA 97,193,0,114,31,254,127,252,96
530 DATA 255,254,56,198,3,142,255,254,56
540 DATA 127,252,96,114,31,254,97,193,0
550 DATA 241,254,0,1,48,0,1,240,0
560 DATA 1,96,0,1,192,0,15,248,0,0
READY.
```

Adding color to sprites

You can make sprites appear in any of the Commodore's 16 colors. As you saw on the previous page, it is just a matter of inserting a color control number with a POKE statement that colors a specific sprite. The color numbers are the same as those shown on page 32 for use with POKE. The POKE location depends on the number of the sprite. For sprite 0 it is V+39 (as in line 80), for sprite 1 it is V+40, and so on.

SETTING SPRITE COLORS

Sprite colors are controlled individually by a color code POKEd into a VIC color location. This table gives both VIC locations (V=VIC base address) and color codes.

Sprite number	VIC color location	Color	Color code	Color	Color code
0	V+39,	Black	0	Orange	8
1	V+40,	White	1	Brown	9
2	V+41,	Red	2	Light red	10
3	V+42,	Cyan	3	Dark gray	11
4	V+43,	Purple	4	Medium gray	12
5	V+44,	Green	5	Light green	13
6	V+45,	Blue	6	Light blue	14
7	V+46,	Yellow	7	Light gray	15

Positioning and moving sprites

If you look at the single sprite program again, you will see this line:

```
90 POKE V,50:POKE V+1,100:POKE V+16,0
```

This controls where the sprite appears on the screen. V sets the horizontal position and V+1 the vertical position. Using two single bytes for a sprite allows you to place it in 256 positions vertically and 256 positions horizontally. But there are actually more than 256 positions across the screen – 512 in fact, so a 9th bit is required to completely specify the position of a sprite. Instead of giving an extra byte to each sprite to store this information, all 8 extra bits are combined and stored in a single VIC register byte, V+16. In this extra byte, bit 0 is for sprite 0, bit 1 for sprite 1 and so on.

SPRITE POSITION CONTROLS

The horizontal and vertical position of each sprite is controlled by a separate VIC location.

Sprite number	Horizontal (X) VIC location	Vertical (Y) VIC location
0	V+0,	V+1,
1	V+2,	V+3,
2	V+4,	V+5,
3	V+6,	V+7,
4	V+8,	V+9,
5	V+10,	V+11,
6	V+12,	V+13,
7	V+14,	V+15,

All horizontal positions outside 24 to 343 and vertical positions outside 30 to 229 are off the screen. The position numbers are arranged in this way so that sprites can be scrolled smoothly on and off the screen in any direction. This does mean that positioning a sprite on the screen so that it fits a background can be rather tricky – an outline of how the sprite co-ordinates work is given on page 59.

Once you know how to position a sprite, it is an easy matter to change this position with a FOR ... NEXT loop. All that remains is to turn the sprite on. In the single sprite program that is done by line 100. Only one bit is needed to turn a sprite on and off, and again, the separate bits for all 8 sprites are combined and put in one VIC register, V+21. For example, if you had 8 sprites, and you wanted some to be off and some to be on at one point in a program, you would work out the POKE number by adding together bits like this:

TURNING SPRITES ON AND OFF

Sprite number	128	64	32	16	8	4	2	1
Bit value	0	1	0	0	1	1	0	1
Status	OFF	ON	OFF	OFF	ON	ON	OFF	ON

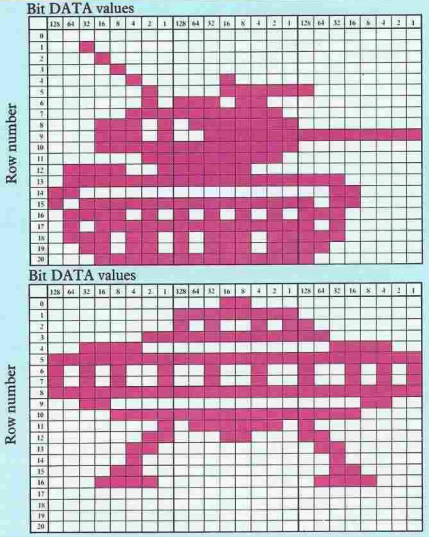
Total byte value =VIC+21,77

By the same technique, you can expand sprites both horizontally and vertically, to twice their original dimensions in both directions. V+29 holds the bits that instruct horizontal expansion, while V+23 controls vertical expansion. You can see from this that line 110 in the single sprite program expands sprite 0 (which has a bit value of 1) in both directions.

A two-sprite program with animation

To show you how to produce more than one sprite simultaneously, here is a program that creates two sprites. First, the sprites are drawn out on grids:

DOUBLE SPRITE PROGRAM DESIGNS



This time there needs to be twice as much DATA, coding 128 bytes altogether with two redundant bytes at the end of each section. Separate instructions are also needed for positioning and coloring the two sprites. Horizontal and vertical enlargements are carried out by putting a value of 3 into V+23 and V+29. The program also makes the two sprites move at different speeds:

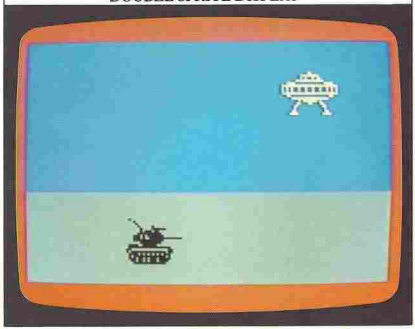
DOUBLE SPRITE PROGRAM

```
LIST -190

10 PRINT CHR$(147);CHR$(19);
20 POKE 53280,2:POKE 53281,12
30 PRINT CHR$(18);CHR$(31);
40 V=53248:FOR C=1 TO 15
50 PRINT " ";:NEXT C
60 FOR C=0 TO 127:READ BYTE
70 POKE 832+C,BYTE:NEXT C
80 X0=0:Y0=150:X1=300:Y1=75
90 D0=1:D1=-4:POKE V+21,3
100 POKE 2040,13:POKE 2041,14
110 POKE V+39,9:POKE V+40,7
120 POKE V+23,3:POKE V+29,3
130 T0=INT(X0/256):T1=INT(X1/256)
140 POKE V+16,T0+2*T1
150 POKE V,X0-T0*256:POKE V+1,Y0
160 POKE V+2,X1-T1*256:POKE V+3,Y1
170 IF X0>350 THEN X0=0
180 IF X1<30 THEN D1=-D1
190 IF X1<30 THEN D1=-D1

READY.
```

```
LIST 200-

200 X0=X0+D0:X1=X1+D1:GOTO 120
210 POKE V+23,3:POKE V+29,3
400 DATA 0,0,0,64,0,0,16,0,0
410 DATA 8,0,0,4,16,0,2,31,128
420 DATA 2,255,0,7,254,0,29,127,0
430 DATA 29,63,255,31,255,0,3,254,0
440 DATA 121,248,0,127,255,224,192,0,48
450 DATA 191,255,176,122,178,224,95,255,
160
460 DATA 118,187,224,54,187,192,31,255,1
28,0
470 DATA 0,24,0,0,255,0,0,165,128
480 DATA 3,255,192,60,0,60,255,255,255
490 DATA 169,36,149,169,36,149,255,255,2
55
500 DATA 48,0,12,15,255,240,1,126,128
510 DATA 3,24,192,6,0,96,4,0,32
520 DATA 12,0,48,24,0,24,0,0,0
530 DATA 0,0,0,0,0,0,0,0,0
READY.
```

Line 80 controls the sprites' initial positions and the variables D0 and D1 in line 90 control the individual directions and speeds of the sprites:

DOUBLE SPRITE DISPLAY



SOUND AND SPECIAL EFFECTS

The Commodore is equipped with one of the most sophisticated sound synthesizer facilities available on any home microcomputer. Sound can make all the difference to your programs, whether it is just a simple bleep to prompt you to enter an INPUT, or a series of realistic sound effects which add excitement and interest to your games. The Commodore does not have a single command like SOUND, but instead uses the multi-purpose command POKE.

The SID chip

All the sounds on the Commodore are created by a single integrated circuit or chip known as the SID (Sound Interface Device). The SID chip contains all the circuitry needed to provide three separate sound channels, each of which can be used to produce musical notes, noises or sound effects. The computer communicates with the SID chip via a block of 29 memory locations, starting at location 54272.

In all the following sound programs the 29 memory locations are referred to as S+N, where S is 54272 (the base address) and N is the SID register number from 0 to 28. For example, memory location 54296, whose main function is as the SID chip master volume control, is shown as S+24. This is the same system as you saw with sprite programming. The simpler numbering is easier to remember, and you are also less likely to make mistakes in entering your programs from a listing.

Simple sound programs

To program sound you need to POKE values for master volume, volume change or "envelope", frequency and waveform. All these details have to be sent to the SID chip. Here to start off is a program which produces the sound of a siren – a loop made up of two notes:

SIREN PROGRAM

```
LIST

300 S=54272 : POKE S+24,15
310 POKE S+5,0 : POKE S+6,240
320 FOR K=1 TO 4
330 POKE S+4,33
340 POKE S,0 : POKE S+1,40
350 FOR T=1 TO 500 : NEXT
360 POKE S,0 : POKE S+1,30
370 FOR T=1 TO 500 : NEXT
380 POKE S+4,32
390 NEXT K
READY.
```

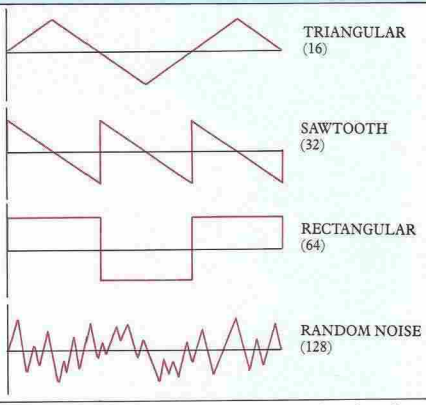
(If you RUN this program and no sound is produced, check the volume control on your television or monitor.)

Line 300 sets the variable S to 54272, and then sets S+24, the master volume, to maximum. The range of volume values runs from 0 (off) to 15 (maximum). Line 310 sets up something called the envelope shape. This controls the way that a sound's volume changes as the note progresses from start to finish. Real sounds don't simply start then stop; they grow and then fade, and the envelope controls these changes. The first half of envelope shape can have any value from 0 to 15 – the higher the number, the slower the change.

The SID chip is also capable of producing sounds with four different waveform shapes, any of which you can select.

SOUND WAVEFORM SHAPES

A waveform shape is selected by POKEing the relevant value into a channel waveform location.



Each of these waveforms is given a numeric value. One of these values needs to be POKEd into the waveform and channel location of each channel you wish to sound, to tell the SID which waveform to play for that channel. Lines 330 and 380 POKE values into this location. Line 330 starts the channel off with a sawtooth waveform and line 380 stops it. You should always make sure that the start values are one greater than the chosen waveform values.

To hear the effect of changing waveforms try modifying the siren program by typing in the following two lines which change the waveform from a sawtooth wave (32) to a triangular wave (16):

```
330 POKE S+4,17
380 POKE S+4,16
```

The frequency or pitch of the two tones of the siren is controlled by lines 340 and 360. Note that you need two POKEs for each pitch. You will find out more about frequency values in music programming on the next two pages. Lines 350 and 370 control the durations of the notes by using simple delay loops.

Programming sound effects

With the random noise waveform you can program a range of sound effects:

GUNFIRE PROGRAM

```
LIST

10 S=54272 : POKE S+24,15
20 FOR K=1 TO 10
30 POKE S+4,129
40 POKE S+5,6 : POKE S+6,89
50 POKE S,0 : POKE S+1,50
60 FOR T=1 TO 65 : NEXT
70 POKE S+4,128
80 NEXT K
READY.
```

This program produces the sound of a volley of machine-gun fire. The envelope shape for each shot is controlled by line 40 which is set so that the volume of the random noise rises quickly to its maximum level and then decays away fairly fast to about half volume before slowly dying away altogether. Now try the following program:

LASER CANNON PROGRAM

```
LIST

100 S=54272 : POKE S+24,15
110 POKE S+5,0 : POKE S+6,249
120 POKE S+4,129
130 FOR C=0 TO 8
140 POKE S,0 : POKE S+1,44-C*4
150 FOR T=1 TO 30 : NEXT
160 NEXT C
170 POKE S+4,128
READY.
```

This produces the sound of a laser cannon firing and shows the effect of changing the frequency of the sound rapidly, from high to low, while it is playing. This effect is created by line 140 which changes the frequency within the FOR ... NEXT loop.

Finally, here are two programs that produce very different effects. The first is a piercing electronic bird call, while the second is the sound of an engine slowly coming to a halt (press RUN/STOP and RESTORE when you've heard enough of them!):

BIRD CALL PROGRAM

```
LIST

10 S=54272:POKE S+24,15
20 POKE S+5,0:POKE S+6,240
30 K=1
40 POKE S+4,65:POKE S+3,8
50 POKE S,0:POKE S+1,100
60 FOR T=1 TO 60:NEXT
70 POKE S+4,65:FOR C=0 TO 6
80 POKE S+1,C*15+150
90 NEXT C:POKE S+4,0
100 FOR T=1 TO 200:NEXT
110 K=K+1:IF K=2 THEN 70
120 FOR T=1 TO 250:NEXT:GOTO 10
READY.
```

ENGINE PROGRAM

```
LIST

10 Z=1
20 S=54272 : POKE S+24,10
30 POKE S+5,9 : POKE S+4,129
40 POKE S,10 : POKE S+1,20
50 FOR T=1 TO Z:NEXT
60 POKE S+5,9 : POKE S+4,17
70 POKE S,17 : POKE S+1,37
80 POKE S+4,0
90 FOR T=1 TO 10:NEXT
100 Z=1.2*Z
110 GOTO 20
READY.
```

SOUND MEMORY LOCATIONS

To specify a sound, the following controls for each channel must be POKEd into the SID chip.

Function	Channel 1	2	3
Note frequency (low value)	S+0,	S+7,	S+14,
Note frequency (high value)	S+1,	S+8,	S+15,
Square wave pulse width (low value)	S+2,	S+9,	S+16,
Square wave pulse width (high value)	S+3,	S+10,	S+17,
Waveform/main channel control	S+4,	S+11,	S+18,
Envelope shape (attack/decay)	S+5,	S+12,	S+19,
Envelope shape (sustain/release)	S+6,	S+13,	S+20,
Filter mode/master volume	S+24,	S+24,	S+24,

NOTES, CHORDS AND MUSIC

All the sound programs you have tried so far have been created on channel 1. You could have just as easily chosen channels 2 or 3. But the Commodore is not limited to playing just one channel at a time. With the right program you can make it play chords – a number of simultaneous notes – and these can be put together to produce harmonies.

To play a single sequence of notes on one channel all you have to do is enter a list of lines that POKEs note values into the frequency control register pair. Here is an example program for channel 1:

CHANNEL 1 TUNE PROGRAM

```
LIST

100 S=54272 : POKE S+24,15
110 POKE S+5,0 : POKE S+6,160
120 POKE S+4,33
130 POKE S,195 : POKE S+1,16
140 FOR T=1 TO 160 : NEXT T
150 POKE S+4,32
160 FOR T=1 TO 20 : NEXT T
170 POKE S+4,33
180 POKE S,195 : POKE S+1,16
190 FOR T=1 TO 80 : NEXT T
200 POKE S,209 : POKE S+1,18
210 FOR T=1 TO 320 : NEXT T
220 POKE S,195 : POKE S+1,16
230 FOR T=1 TO 320 : NEXT T
240 POKE S,96 : POKE S+1,22
250 FOR T=1 TO 320 : NEXT T
260 POKE S,30 : POKE S+1,21
270 FOR T=1 TO 320 : NEXT T
280 POKE S+4,32
READY.
```

This method works well, but you can quickly see that it would take several screens of listing for a tune of any length. A better solution is to store the frequency or pitch values in DATA statements and have a short program to READ and play the notes. Here's the same tune, with slight modifications:

TUNE PROGRAM WITH DATA

```
LIST

300 S=54272 : POKE S+24,15
310 POKE S+5,0 : POKE S+6,160
320 READ L,H,D
330 IF H=0 THEN FOR T=1 TO 20 : NEXT T :
  GOTO 320
340 IF H<0 THEN END
350 POKE S+0,L : POKE S+1,H
360 POKE S+4,33
370 FOR T=1 TO D : NEXT T
380 POKE S+4,32
390 GOTO 320
400 DATA 195,16,160,0,0,0,195,16,80
410 DATA 209,18,320,195,16,320
420 DATA 96,22,320,30,21,320,0,-1,0
READY.
```

There are three DATA values in this program for each note, two for the frequency and one for the duration. If the "high" frequency is 0 then the program will produce a silence and if it is negative then this will END the program, simply terminating it without PRINTing a line number.

Once you have entered this program, playing any tune is just a matter of carefully converting each note in the tune into its equivalent high- and low-frequency values and putting them along with a duration into the DATA statements.

How to play chords

With channel 1, you can only play single notes, but by using all three channels you can play chords:

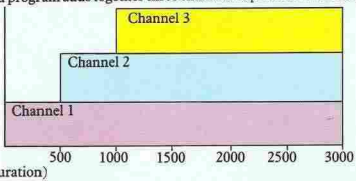
SIMPLE CHORDS

```
LIST

10 S=54272 : POKE S+24,15
20 POKE S+5,0 : POKE S+6,160
30 POKE S+12,0 : POKE S+13,160
40 POKE S+19,0 : POKE S+20,160
50 POKE S+4,17
60 POKE S+0,38 : POKE S+1,8
70 FOR T=1 TO 500 : NEXT T
80 POKE S+11,17
90 POKE S+7,143 : POKE S+8,10
100 FOR T=1 TO 500 : NEXT T
110 POKE S+18,17
120 POKE S+14,143 : POKE S+15,12
130 FOR T=1 TO 2000 : NEXT T
140 POKE S+4,16 : POKE S+11,16 : POKE S+
18,16
READY.
```

Try RUNning the program and listen to the result. The chord builds up to three notes playing at the same time, the maximum number that can be played together. This enables you to play any tune where up to three notes need to be played simultaneously.

ADDING TOGETHER CHANNELS

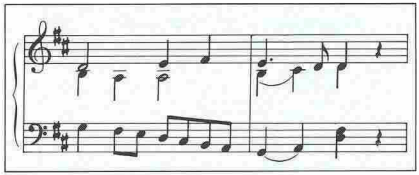
The chord program adds together three channels to produce a harmony.



Programming sheet music

The next program is a chord version of the program to READ notes from DATA statements. The musical

score below shows the notes that have been written into the program. When you are trying this, remember that "tied" notes are continuous, so that if they are of the same pitch you can treat them as one note by combining their durations.



How the music program works

An important fact to remember when programming music on the Commodore is that once the SID chip has been instructed to produce a sound on a particular channel, it will continue to do so until the sound is turned off, or until a new sound is programmed on that channel. In the music program here, the DATA values again control frequency and duration, but this time they control channel as well. If you look at line 130, you will see a series of nine numbers. The first of each group of three is READ as a channel number, and this is followed by the high- and low-frequency values for a note in the first chord.

If you now look at line 140, you will see that the first figure is 0. Line 70 tells the computer that after READing a zero, it will treat the next number as the length of time during which it stays in a delay loop. In this case the next number is 160, so the computer waits FOR T=1 TO 160 before it goes on to READ the next DATA number, which resets channel 2.

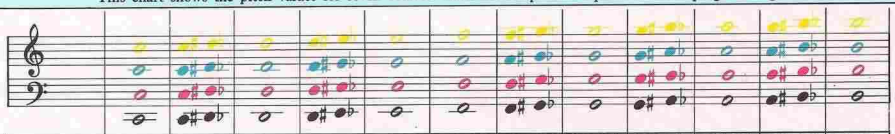
SHEET MUSIC PROGRAM

```
LIST -120

10 S=54272 : POKE S+24,15
20 POKE S+5,0 : POKE S+6,240
30 POKE S+12,0 : POKE S+13,240
40 POKE S+19,0:POKE S+20,240
50 POKE S+4,33 : POKE S+11,33 : POKE S+1
8,33
60 READ C
70 IF C=0 THEN READ D : FOR T=1 TO D : N
EXT T : GOTO 60
80 IF C<0 THEN 120
90 READ L,H
100 POKE S+(C*7)-7,L : POKE S+(C*7)-6,H
110 GOTO 50
120 POKE S+4,32 : POKE S+11,32 : POKE S+
18,32 : END

READY.
```

```
LIST 130-

130 DATA 1,209,18,2,210,15,3,143,12
140 DATA 0,160,2,24,14,3,218,11,0,80
150 DATA 3,143,10,0,80,1,30,21,2,24,14,3
,33,11,3,164,9
160 DATA 0,80,3,225,8,0,80,1,180,23
170 DATA 3,233,7,0,80,1,12,7,0,80
180 DATA 1,30,21,2,210,15,3,71,6,0,160
190 DATA 2,194,17,3,12,7,0,80
200 DATA 1,203,18,0,80,1,218,11,2,24,14
210 DATA 3,104,9,0,320
220 DATA -1
READY.
```

SELECTING A PITCH VALUE

This chart shows the pitch values for seven octaves. Each note requires two pitch values for programming.



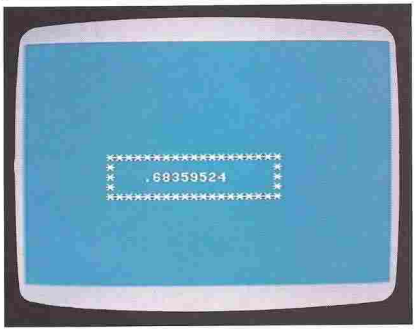
7	134 24	142 18	150 132	159 120	168 243	178 255	189 164	200 235	212 221	225 133	238 238	253 35
6	67 12	71 9	75 66	79 188	84 122	89 127	94 210	100 117	106 110	112 195	119 119	126 146
5	33 134	35 132	37 161	39 222	42 61	44 192	47 105	50 59	53 55	56 97	59 188	63 73
4	16 195	17 194	18 209	19 239	21 30	22 96	23 180	25 29	26 156	28 49	29 222	31 164
3	8 98	8 225	9 104	9 247	10 143	11 48	11 218	12 143	13 78	14 24	14 239	15 210
2	4 49	4 113	4 180	4 252	5 72	5 152	5 237	6 71	6 167	7 12	7 119	7 233
1	2 24	2 56	2 90	2 126	2 164	2 204	2 247	3 36	3 33	3 134	3 188	3 245
0	1 12	1 28	1 45	1 63	1 82	1 102	1 123	1 146	1 170	1 195	1 222	1 250
NOTE	C	C# Db	D	D# Eb	E	F	F# Gb	G	G# Ab	A	A# Bb	B# Cb

UNPREDICTABLE PROGRAMS

Although computers generally work with precise information, doing exactly what you tell them to do, most computer games are based to some extent on chance. If you want to make something happen at an unpredictable time, or if dice are to be thrown or coins tossed, you can't tell the computer what results to produce every time or the element of chance would disappear. The way to build chance into a program is to use RND. You will already have come across this command – it was used to produce a series of random numbers for example in the math test program on pages 30–31. RND, as you have probably guessed, stands for RaNDom. It allows you to generate random numbers up to a maximum that you can set. You can then use these numbers to produce unpredictable sequences. The following program uses RND to PRINT a series of random numbers. The numbers selected by the program are all decimal fractions:

RANDOM NUMBER PROGRAM

```
LIST

10 PRINT CHR$(147);CHR$(5): PRINT : PRIN
T
20 POKE 53280,4:POKE 53281,6
30 FOR C=9 TO 27 : POKE 214,10 : PRINT
40 PRINT TAB(C);"*" : POKE 214,14
50 PRINT : PRINT TAB(C);"*"
60 NEXT C
70 FOR R=11 TO 13 : POKE 214,R : PRINT
80 PRINT TAB(9);"*";TAB(27);"*"
90 NEXT R
100 POKE 214,12 : PRINT : POKE 211,12
110 PRINT RND(0);
120 FOR T=1 TO 1000 : NEXT
130 POKE 214,12 : PRINT : POKE 211,10
140 PRINT "                    ";
150 GOTO 100
READY.
```



This uses RND(0) in line 110 to generate random numbers between 0 and 0.999999999, while lines 30 to 90 set up a border of asterisks to frame the numbers. Very small numbers include the E symbol that you came across on page 19. Normally, as each new number is PRINTed, it automatically erases the last number – simply by PRINTing on top of it. However, when something like E-4 appears, it is not automatically erased. Line 140 has therefore been written into the program to take care of that.

Although RND is called the random function, this is not strictly true. More correctly, it is known as a "pseudorandom" function, one which produces results according to set patterns. Which particular pattern is produced by RND(0) is determined by the time that has elapsed since the computer was switched on. Since there are so many different possible values for this, the numbers produced are for most purposes completely random.

Producing random whole numbers

If you now replace line 110 with:

```
110 PRINT INT(RND(0)*10);
```

and RUN the program again, you will notice an immediate change in the display. The numbers are no longer decimal fractions, in fact there's no decimal point at all. Instead, the program is generating whole numbers between 0 and 9 inclusive – a much more useful result for programs. INT, which you came across on page 29, rounds the random number produced down to the nearest whole number or integer. By using RND, it is quite easy to get the Commodore to simulate throwing dice or tossing coins. Here is a program which selects random numbers to imitate coin throws:

COIN TOSS PROGRAM

```
LIST

10 PRINT CHR$(147)
20 PRINT "HEADS-TAILS"
30 PRINT : PRINT
40 A=INT(RND(0)*2)+1
50 IF A=1 THEN COIN$="HEADS" : GOTO 70
60 COIN$="TAILS"
70 PRINT COIN$,
80 FOR Q=1 TO 200 : NEXT
90 GOTO 40
READY.
```

As a tossed coin can have only one of two values – heads or tails – line 40 produces a random number that is either 1 or 2. Heads are represented by 1 and tails by 2. COIN$ represents the result of the current throw. If A=1, then COIN$ becomes HEADS and line 60 is by-passed. If A=2, the computer decides that the condition in line 50 is not true, and so it goes on to set COIN$ to TAILS in line 60. The FOR ... NEXT loop using the variable Q, in line 80, produces a short pause between each throw:

COIN TOSS DISPLAY

```
HEADS-TAILS

TAILS     HEADS     TAILS     TAILS
HEADS     HEADS     TAILS     TAILS
HEADS     HEADS     TAILS     TAILS
TAILS     TAILS     TAILS     TAILS
HEADS     HEADS     HEADS     TAILS
TAILS     HEADS     TAILS     HEADS
HEADS     HEADS     TAILS     HEADS
HEADS     HEADS     TAILS     HEADS
TAILS     TAILS     HEADS     HEADS
HEADS     TAILS     TAILS     TAILS
HEADS     TAILS     HEADS     HEADS
HEADS     TAILS     TAILS     TAILS
HEADS     HEADS     TAILS     TAILS
TAILS     TAILS     TAILS     TAILS
TAILS     TAILS     HEADS     HEADS
TAILS     HEADS     TAILS     HEADS
HEADS     TAILS     TAILS     HEADS
```

How random is a random program?

It is possible to write a program that will show you just how random RND is. If you use RND to toss an electronic "coin" 100 times, you should get roughly 50 heads and 50 tails each time. You can actually test to see if this is true. Key in this program by amending the coin toss program. When you RUN it, the computer will PRINT totals showing how close to 50:50 the heads and tails are each time the program is carried out:

RANDOM TEST PROGRAM

```
LIST

10 PRINT CHR$(147) : N=0 : H=0 : T=0
20 PRINT TAB(13);"HEADS-TAILS"
30 FOR K=1 TO 100
40 A=INT(RND(0)*2)+1 : N=N+1
50 IF A=1 THEN COIN$="HEADS" : GOTO 70
60 COIN$="TAILS"
70 POKE 214,5 : PRINT : POKE 211,10
80 PRINT "THIS THROW - ";COIN$ : PRINT
90 PRINT TAB(5);"HEADS";TAB(28);"TAILS"
100 IF A=1 THEN H=H+1 : GOTO 120
110 T=T+1
120 POKE 214,9 : PRINT : POKE 211,5
130 PRINT TAB(6);H;TAB(29);T : PRINT
140 PRINT TAB(10);"TOTAL THROWS =";N
150 FOR Z=1 TO 500 : NEXT
160 NEXT K
READY.
```

Using random effects in graphics programs

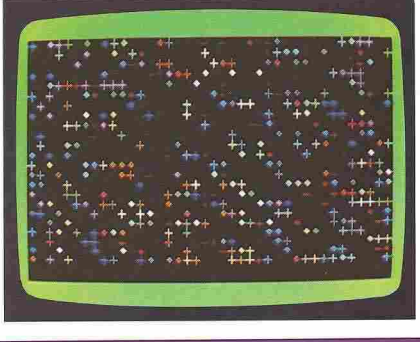
You can produce some fascinating effects with RND by incorporating it in graphics programs, so that the computer is instructed to draw random graphics characters at random positions on the screen. If you then make the computer repeat this a number of times by using FOR ... NEXT, you can build up a complete graphics pattern which will be different every time the program is RUN. This program uses RND in this way. Line 40 chooses a random text color, and lines 50 and 60 pick two random X and Y co-ordinates:

RANDOM GRAPHICS PROGRAM

```
LIST
10 PRINT CHR$(147);CHR$(5)
20 POKE 53280,5:POKE 53281,0
30 FOR N=1 TO 1000
40 COL=RND(0)*15
50 X=INT(RND(0)*40)
60 Y=INT(RND(0)*23)
70 POKE 1024+Y*40+X,90+RND(1)*2:POKE 552
96+Y*40+X,COL
80 NEXT N
READY.
```

Line 70 looks more complicated than it really is. Using the formula you saw first on page 36, it POKEs a character and color at the screen position X,Y. This time, the character can be either a diamond or a cross – the selection is made every time by 90+RND(1)*2 which picks the character code. You can use this program as a basis for producing a whole range of different displays by altering character and color settings:

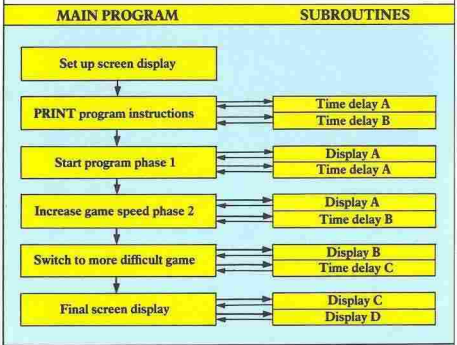
RANDOM GRAPHICS DISPLAY



WRITING SUBROUTINES

You will often want to use the same few lines of a program again and again to carry out the same calculation or to display the same group of characters on the screen. To avoid writing out the same lines time after time (and using up too much of the computer's memory) you could branch off to frequently used sections of the program with GOTO. However, the use of GOTO is frowned on by many programmers. Using it carelessly can turn your programs into untidy mazes that are impossible to understand or debug.

The easiest program to analyze and debug is one that is written methodically in blocks or modules, each of which you can test independently of the other, if problems arise. If you look up the listing of a good games program in a magazine, for example, you will find that it works something like this:



How to use a subroutine

With the Commodore, frequently used blocks of programs can be "set aside" as subroutines. Subroutines are set up using the command GOSUB. This command allows you to branch off from the main program to the subroutine and then return to the main program again. The command looks like this:

```
50 GOSUB 500
```

Here the main program RUNs normally until it reaches line 50, which makes the computer jump to a subroutine at line 500. After it has been through the subroutine, it automatically returns to the main program at line 60 – the one after the line where it left. The subroutine must be ended by the word RETURN. Without it, the computer will not go back to the correct point in the main program.

You can use GOSUB in almost any program where the computer has to repeat an operation. The next program produces a temperature conversion chart, using three types of measurement, Centigrade, Fahrenheit and Kelvin. The subroutine at lines 90 to 110 makes the computer PRINT out a line on the table and then RETURN to line 70. The command END at line 80 stops the main program carrying on into the subroutine. If you miss out END, the computer will reach the RETURN command at line 110. It would then produce an error message because it had encountered a RETURN without its own GOSUB. You will also notice here that the subroutine is actually inside a FOR ... NEXT loop, so it is "called" each time the Centigrade temperature is increased by the NEXT command. There is a new command – STEP – in line 50. This makes the loop increase C in jumps of 10 instead of 1. STEP does not always have to be a whole number (it can even be negative):

TEMPERATURE CONVERSION PROGRAM

```
LIST

10 PRINT CHR$(147);CHR$(31)
20 POKE 53280,8 : POKE 53281,3
30 PRINT : PRINT TAB(6);"C";TAB(18);"F";
TAB(31);"K"
40 PRINT "----------
----------"
50 FOR C=-40 TO 120 STEP 10
60 GOSUB 90
70 NEXT C
80 END
90 PRINT TAB(4);C;TAB(16);C*9/5+32;TAB(2
9);C+273
100 FOR A=1 TO 200 : NEXT A
110 RETURN
READY.
```

```
C          F          K

-40        -40        233
-30        -22        243
-20        -4         253
-10        14         263
0          32         273
10         50         283
20         68         293
30         86         303
40         104        313
50         122        323
60         140        333
70         158        343
80         176        353
90         194        363
100        212        373
110        230        383
120        248        393
```

In this program, the subroutine is not actually saving any space. However, if you extended the program to carry out other functions, the subroutine could be "called" again as often as you wanted – saving both space and memory.

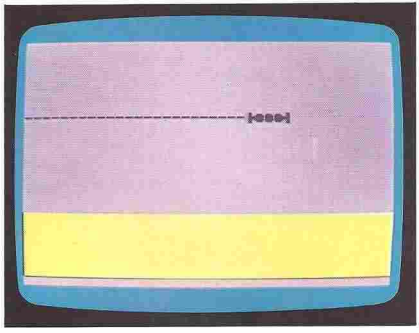
Setting up "menu" displays with GOSUB

In many programs, you are initially given a "menu" or choice of options to select. This choice is often programmed by using GOSUB. When you enter your selection, the program goes to the appropriate subroutine and sets up the display for the game or function that you have picked.

Here is a simple listing that shows how you can do this. The program can set up either of two basic displays. The displays are produced by a subroutine – which colors are used depends on your INPUT following line 20. In this program, a keyboard character group moves across the screen. If you were using it in a real games program, you could call the subroutine as often as you wanted:

MENU PROGRAM

```
LIST

10 PRINT CHR$(147)
20 INPUT "1 OR 2";A
30 PRINT CHR$(147) : IF A=1 THEN G=158 :
 S=4 : C=144 : GOSUB 200
40 IF A=2 THEN G=28 : S=5 : C=28 : GOSUB
 200
50 FOR X=0 TO 34
60 POKE 214,6 : PRINT : POKE 211,X : PRI
NT "-■■■■■";
70 FOR M=1 TO 50: NEXT M
80 NEXT X
90 POKE 214,6:PRINT:POKE 211,34:PRINT"
  "
100 GOTO 50
200 POKE 53280,6: POKE 53281,S
210 POKE 214,16 : PRINT
220 PRINT CHR$(G) ; FOR Y=1 TO 280
230 PRINT "  ■"; : NEXT Y
240 PRINT CHR$(C); : RETURN
READY.
```



GOSUB is a particularly useful command for many games programs in which a tune is played over and over again. Instead of repeatedly typing the listing for the tune, you can simply write it once, and then enter a GOSUB command, followed by the appropriate line number, each time you want to repeat the tune.

Using GOSUB with animation

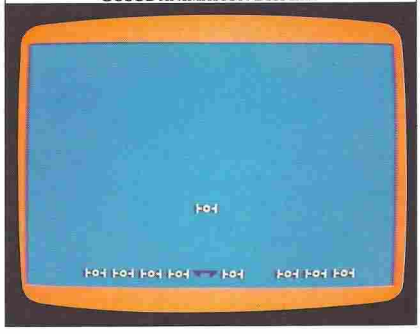
As you will have discovered when learning about simple graphics, writing programs that involve animation can be very time-consuming, particularly if you want several characters to move. GOSUB is particularly useful in programming animation:

GOSUB ANIMATION PROGRAM

```
LIST

10 PRINT CHR$(147)
20 POKE 53280,2 : POKE 53281,6
30 POKE 214,23 : PRINT : POKE 211,18
40 PRINT CHR$(144);CHR$(127);CHR$(184);C
HR$(169)
50 X=INT(RND(1)*13)*3
60 PRINT CHR$(158) : FOR Y=1 TO 22
70 POKE 214,Y-1 : PRINT : POKE 211,X
80 PRINT "  " : POKE 211,X
90 PRINT CHR$(171);CHR$(119);CHR$(179);
100 IF X=18 THEN IF Y=22 THEN GOSUB 500
110 FOR T=1 TO 5:NEXT T
120 NEXT Y : GOTO 50
500 FOR T=1 TO 20
510 POKE 214,22 : PRINT : POKE 211,18
520 PRINT CHR$(28);CHR$(127);CHR$(184);C
HR$(169);
530 POKE 214,22 : PRINT : POKE 211,18
540 PRINT CHR$(144);CHR$(127);CHR$(184);
CHR$(169);
550 NEXT T : RETURN
READY.
```

Here is a program which PRINTs a target – three graphics symbols – on the ground, and which then PRINTs a succession of aliens falling from random points at the top of the screen. If one of the aliens then hits the target, line 100 directs the computer to the subroutine at line 500, producing a series of flashing colors before the program resumes again:

GOSUB ANIMATION DISPLAY



HINTS AND TIPS

When you are learning to program your Commodore, you will probably have come across a number of ways of improving your technique by trial and error. However, there are some methods of saving time or sorting out problems which, although simple and effective, are not necessarily obvious. On these two pages you will find some "tricks" which will help you to produce programs that are well organized and bug-free.

Using REM as a marker or mask

Because the command REM makes the computer ignore anything that follows it in a line, it can be used in labeling and testing parts of a program. On page 20 you saw how REM can be used in the first line of a program to show you what the program does. You can also use REM lines throughout a program to remind you what each of the parts does, and the longer a program is, the more helpful this becomes.

However, when a program gets really long, it is sometimes difficult to pick out REM lines among all the others. One way you can draw attention to them is by following REM with some symbols which clearly stand out from the rest of the program. Here is one way of doing this:

PROGRAM LABELED WITH **REM**

```
LIST

 2 REM ******************************
 4 REM       TEMPERATURE CHART
 6 REM ******************************
 8 REM
 10 PRINT CHR$(147);CHR$(31)
 20 POKE 53280,10 : POKE 53281,3
 30 PRINT "   PRINT TAB(6);"C";TAB(18);"F";
TAB(31);"K"
 40 PRINT "----------"
----------"
 50 FOR C=-40 TO 120 STEP 10
 60 GOSUB 90
 70 NEXT C
 80 END
 90 PRINT TAB(4);C;TAB(16);C*9/5+32;TAB(2
9);C+273
 100 FOR A=1 TO 200 : NEXT A
 110 RETURN
READY.
```

When you read through this program, the REM lines are visible at a glance.

REM also has a use in program development. You will often want to test a program to see what happens if certain lines are left out. This may be because part of the program takes a long time to RUN, or produces a sound that you don't want to hear time and time again!

You can skip part of a program by using GOTO or RUN followed by a line number, but this won't help if you just want to miss out a few lines in the middle. The way to deal with this problem without resorting to deleting and losing the lines altogether is to insert a REM command at the beginning of each line you want to skip. This will mask or "disable" the lines, as the computer will ignore all the commands following each REM. Here is a program in which this has been done:

LINES MASKED WITH **REM**

```
LIST

 10 POKE 53280,2:POKE 53281,2
 20 PRINT CHR$(147);CHR$(5)
 30 REM PRINT " A";" A*2";" A*3"
 40 REM PRINT "----------"
-----------"
 50 FOR N=1 TO 10
 60 PRINT N,N*N,N*N*N
 70 NEXT N
READY.
```

How to check nested loops

When you use a number of loops in a program, it is easy to get the loops tangled up so that the program does not produce the result you want. But there is an easy way of checking if loops are "nested" without overlapping.

You do this by connecting up the beginning and ending of each loop with a line. Here is a program with nested loops, showing how these connecting lines should fit inside each other. There are three FOR ... NEXT loops, two contained within the main loop between lines 20 and 90:

PROGRAM WITH NESTED LOOPS

```
LIST

 10 PRINT CHR$(147)
 20 FOR Y=0 TO 20
 30 FOR X=0 TO 38
 40 PRINT CHR$(144);"■";
 50 POKE 55296+Y*40+X,X
 60 NEXT X
 70 FOR T=1 TO 50
 80 NEXT T
 90 NEXT Y
READY.
```

When you have connected up every FOR with its NEXT, you should find that none of your lines overlaps any other. If any does, you have wrongly nested loops, and the chances are that your program will not work correctly. Of course you can't draw lines on the screen itself, but this method can be used on a program layout or a printed listing.

Useful debugging techniques

Although the Commodore has a large repertoire of error messages which will alert you to any incorrect lines in a program, often a program will RUN without any hitches, only to produce a result entirely different to the one you had in mind. How then do you go about finding the source of the problem?

As you have just seen, you can use REMs to mask parts of a program, or you can link loops to check that they are nested properly. But if that doesn't help, you can often track down the problem by giving each variable in a program one set value, instead of allowing it to go through many.

Imagine that you have a graphics program which uses the command RND to produce a display which is built up by looping. If it does not work in the way you expect, you can take out the RND, and instead use a number. You can then work out what effect this number should have when the program is RUN. Now take out the lines that start and terminate the loop (you can use REM for this). If the result of a single RUN through is not what you predicted, the display should give you some idea of where your program is going "wrong":

PROGRAM EDITED FOR TESTING

```
LIST

10 PRINT CHR$(147);CHR$(5)
20 POKE 53280,5:POKE 53281,0
30 REM FOR N=1 TO 1000
40 COL=5      REM COL=RND(0)*15
50 X=20       REM X=INT(RND(0)*40)
60 Y=15:      REM Y=INT(RND(0)*23)
70 POKE 1024+Y*40+X,90+RND(1)*2:POKE 552
96+Y*40+X,COL
80 REM NEXT N
READY.
```

Above is the random graphics program from page 53, edited so that the random variables in lines 40 to 60 are fixed. The original lines are still kept in, but are disabled by REMs. The loop between lines 30 and 80 is also disabled by a pair of REMs so the program only PRINTs once.

If the program is RUN, you can check whether or not the program has done what was expected, and if not, it is now much easier to work backward to the source of the problem. You can use this technique in any program which uses variables. By substituting a single value for each variable, you can check your expected result with the result when the program is RUN.

Don't forget that the STOP key can be very helpful in telling you how far the computer has got through a program. If you RUN a program which either seems to do nothing, or gets stuck at a certain point, the STOP key will tell you where the hold-up lies. If you then LIST the program, you will often be able to identify the problem with the line identified by STOP and correct it so that the program works.

How to recover lost programs

Finally, if you do much programming sooner or later you will probably lose a listing by accidentally typing NEW before you SAVE it. However, you don't have to start programming all over again. The following screen shows a short sequence of direct commands for program recovery (CLR means press SHIFT+CLR):

CANCELING NEW

```
POKE 2050,8
SYS 42291
POKE 45,PEEK (174)
POKE 46,PEEK (175)
POKE 47,PEEK (174)
POKE 48,PEEK (175)
POKE 49,PEEK (174)
POKE 50,PEEK (175)
CLR
LIST
```

This works because when you type NEW, BASIC does not actually erase the program listing from memory. All that happens is that several pointers that tell BASIC where the program starts and where to store all its variables are reset to values that make BASIC think that there's no program in memory. Your old program actually remains in RAM until you start a new one.

The first two lines on the screen sort out where in memory the deleted program ends. This is stored in memory at locations 174 and 175. The rest of the commands copy this address back into the pointers that were lost when NEW was keyed in. The pointers at locations 45/46 and 47/48 are used to tell BASIC where its list of variables starts and the pointer at locations 49/50 is used by BASIC to mark the position in memory where its string space ends.

HOW TO KEEP YOUR PROGRAMS

Whatever you type into your Commodore is only stored in the computer's memory for as long as the power is on. When you switch off, your program disappears. Obviously, you can't type in every program you want to use each time you switch on. Fortunately the Commodore offers two ways to store programs by using either a cassette recorder or disk drive.

The Commodore cassette recorder is a very simple machine to use because it is designed specifically for storing Commodore programs, and all its controls are permanently set to receive the computer's signals. The disk system allows you to carry out more complex data handling, but at the simplest level, it uses just the same commands as the cassette recorder.

Program storage commands

Programs are recorded onto tape or disk and played back again using the commands SAVE and LOAD. You can try these out with any program in this book. Type a listing into the computer and then RUN the program to make sure that it contains no typing errors. Then decide on a filename for the program not more than 16 characters long. Now type in SAVE "FILENAME", using the name you have chosen. If you are using a disk, type SAVE "FILENAME",8. When you press RETURN, the process begins:

CASSETTE **SAVE** MESSAGE

```
LIST

10 POKE 53280,6:POKE 53281,7
20 PRINT CHR$(147)
30 FOR X=1 TO 20
40 PRINT X,X*2,X+3
50 NEXT X
READY.


SAVE "FOR-NEXT"
PRESS RECORD & PLAY ON TAPE
```

With a tape system, SAVEing starts when you press RECORD and PLAY. With a disk system, you do not have to do anything further after keying in the SAVE command. When the computer has SAVEd the program, the READY message will appear. Your program should now be stored.

How to check tapes and disks

It is always a good idea to check that the program you have just SAVEd is in fact stored correctly. To do this you type the command VERIFY followed by the filename (and ,8 if you are using a disk system) and press RETURN. If you are using tapes, when the computer finds a program on the tape it will PRINT the filename. To get the computer to VERIFY the program, press the C= key (this is not necessary for disk systems). Now the computer will compare the SAVEd program with the one in its memory. If there are any differences, a VERIFY ERROR message will show you that the SAVEd program has been stored incorrectly.

Playing programs back

If you now type NEW, you can try recalling a program. Type in LOAD "FILENAME" (again add ,8 for disks). The program will now be LOADed into the computer ready for RUNning. With a disk, the drive will automatically move to the blocks where the program has been stored. With a tape system, it helps to use the counter to move to the right part of the tape first:

LOADING A TAPED PROGRAM

```
LOAD"FOR-NEXT"

SEARCHING FOR FOR-NEXT
LOADING
READY.


LIST

10 POKE 53280,6:POKE 53281,7
20 PRINT CHR$(147)
30 FOR X=1 TO 20
40 PRINT X,X*2,X+3
50 NEXT X
READY.
```

If you type in LOAD "$",8 with a disk system, the computer will LOAD the disk's directory. Typing LIST will then enable you to see the filenames of all the programs on the disk.

LOADing from inside a program

There is a way by which you can make one program RUN another (this is known as "chaining"). If you use the LOAD command as part of a program line, the computer will LOAD the program specified and then go on to RUN it. With this technique you can link a number of programs together. However, if you do this you should remember that values of variables in any previous programs will remain in memory. This does not happen when LOAD is used outside a program.

SPRITE GRIDS

Commodore sprites are each composed of 21 rows of 24 pixels, each pixel being controlled by a single bit in a POKE statement. You can use the grid below to design your own sprites and work out their POKE values. These can then be entered as DATA in a program. After pencilling in a design on the grid, add up the bit values for each group of eight pixels on a row. You can then record the totals in the columns on the right.

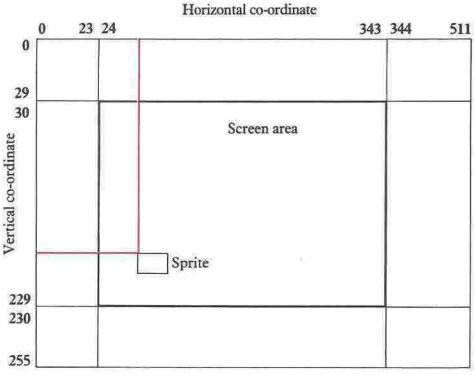
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

DATA values		

Positioning sprites

Sprite position co-ordinates can have any value from 0–255 vertically or 0–511 horizontally. However, only a part of this range is actually visible on the screen. The diagram on the right shows how these co-ordinates relate to the screen. The central panel is the visible area on the screen. This extends from 24 to 343 horizontally and 30 to 229 vertically. This means that you can move sprites on and off the screen smoothly.

Because vertical position runs from 0 to 255, it can be controlled by a single byte of information. Horizontal position, on the other hand, needs two bytes if the whole of the range from 0 to 511 (=2×255) is to be used. If you want a sprite to continue moving past horizontal position 255, the V+16 location must be turned on for the sprite concerned.



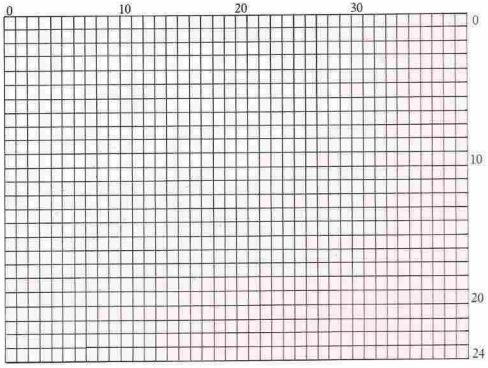
SCREEN MEMORY CODES

At text resolution, every point on the Commodore screen can be accessed using the POKE command. This allows you to put characters in any color onto the screen at specified positions. Two separate POKE commands are needed – one to specify the character that is to appear at a screen position, and another to set its color. The grid below shows how the screen memory and color memory locations are selected.

Screen memory and color memory

To find out the two memory locations of any point on the screen, first find out the row POKE values. The complete screen and color memory locations can then be determined by adding the column number to each POKE value. Once you have determined these two numbers, you can add a character and color value.

1024	55296
1064	55336
1104	55376
1144	55416
1184	55456
1224	55496
1264	55536
1304	55576
1344	55616
1384	55656
1424	55696
1464	55736
1504	55776
1544	55816
1584	55856
1624	55896
1664	55936
1704	55976
1744	56016
1784	56056
1824	56096
1864	56136
1904	56176
1944	56216
1984	56256



Character POKE values

The chart below shows the POKE values for the Commodore's upper case and lower case characters. Character codes are POKEd together with the screen memory values. Values from 128 to 255 produce the reverse images of the characters shown here.

	0	1	2	3	4	5	6	7	8	9
0	@	A a	B b	C c	D d	E e	F f	G g	H h	I i
10	J j	K k	L l	M m	N n	O o	P p	Q q	R r	S s
20	T t	U u	V v	W w	X x	Y y	Z z	[	£	]
30	↑	←	SPACE	!	"	#	$	%	&	'
40	(	)	*	+	,	−	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?		A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z						SPACE			
100										
110										
120										

ASCII CHARACTER SET

The ASCII character set forms a single sequence of characters and control functions that can be accessed by the command CHR$. The ASCII system provides a standard digital coding for computer characters. The codes from 33 to 127 represent the same characters on almost all microcomputers, while the codes outside this range are used differently by different machines. On the Commodore these control a range of functions like color settings and represent keyboard graphics characters. The ASCII code for each character only uses 7 bits of a byte, leaving room for an eighth bit for "parity checking", or transmission error monitoring.

	0	1	2	3	4	5	6	7	8	9
0						White			Disables SHIFT+C=	Enables SHIFT+C=
10				RETURN	Lower case			Cursor down	RVS ON	CLR HOME
20	INST/DEL								Red	Cursor right
30	Green	Blue	SPACE	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	£	]	↑	←				
100										
110										
120										Orange
130				Function key 1	Function key 2	Function key 3	Function key 4	Function key 5	Function key 6	Function key 7
140	Function key 8	SHIFT + RETURN	Upper case		Black	Cursor up	RVS OFF	CLR HOME	INST/DEL	Brown
150	Light red	Light gray	Medium gray	Light green	Light blue	Dark gray	Purple	Cursor left	Yellow	Cyan
160	SPACE									
170										
180										
190										
200										
210										
220					SPACE					
230										
240										
250										

GLOSSARY

Entries in **bold type** are BASIC keywords.

ASCII
American Standard Code for Information Interchange; the character coding system used by the Commodore.

BASIC
Beginners' All-purpose Symbolic Instruction Code; the most commonly used high-level programming language.

Binary
A counting system used by computers based on only two numbers – 0 and 1.

Bit
A binary digit – 0 or 1.

Byte
A group of eight bits.

Chip
A single package containing a complete electronic circuit. Also called an integrated circuit (IC).

CHR$
Converts an ASCII code into the character it represents.

CPU
Central Processing Unit. Normally contained in a single chip called a microprocessor, this carries out the computer's arithmetic and controls operations in the rest of the computer.

Cursor
A flashing symbol on the screen, showing where the next character will appear.

DATA
The computer treats whatever follows **DATA** as information that may be needed later in the program. Used in conjunction with **READ**.

Debugging
The process of ridding a program of errors or bugs.

END
Halts a program. (See also **STOP**.)

Envelope
The change in amplitude (volume) of a note while it is playing. Envelope shapes are selected with **POKE**.

Filename
A name given to a program or set of data to enable storage and recall on a tape or disk.

Flowchart
A diagrammatic representation of the steps necessary to solve a problem.

FOR ... NEXT
A loop which repeats a sequence of program statements a specified number of times.

GOSUB
Makes the program jump to a subroutine beginning at the line number following the command. The subroutine must always be terminated by **RETURN**.

GOTO
Makes a program jump to the line number following the command.

Hardware
The physical machinery of a computer system, as distinct from the programs (software) that make it do useful work.

IF...THEN
Prompts the computer to take a particular course of action only if the condition specified is detected.

INPUT
Instructs the computer to wait for some data from the keyboard which is then used in a program.

INT
Converts a number with a decimal fraction into a whole number by rounding down.

Interface
The hardware and software connection between a computer and another piece of equipment.

K
Abbreviation of kilobyte (1024 bytes).

LET
Assigns a value to a variable. The use of **LET** is optional on the Commodore.

LIST
Makes the computer display the program currently in its memory.

LOAD
Transfers a program from a tape or disk into the computer's memory. The program is identified by a filename.

Loop
A sequence of program statements which is executed repeatedly or until a specified condition is satisfied.

NEW
Removes a program from the computer's memory.

ON ... GOTO/GOSUB
Makes a program jump to one of a number of statements or subroutines depending on the value of a variable.

PEEK
Reads the numeric value in a specified memory location.

POKE
Puts a numeric value into a specified memory location.

PRINT
Makes whatever follows appear on the screen.

RAM
Random Access Memory (volatile memory). A memory whose contents are erased when the power is switched off. (See also ROM.)

READ
Instructs the computer to take information from a **DATA** statement.

REM
Enables the programmer to add remarks to a program. The computer ignores whatever follows **REM** in a program statement.

RESTORE
Resets the computer to **READ** the first item in a **DATA** list.

RETURN
Terminates a subroutine. (See also **GOSUB**.)

RND
Produces numbers at random within specified limits.

ROM
Read Only Memory (non-volatile memory). A memory which is programmed permanently by the manufacturer and whose contents can only be read by the user's computer.

SAVE
Records a program currently in memory onto a tape or disk. The program is identified by a filename.

SID
Sound Interface Device; the chip used by the Commodore to produce sound.

Software
Computer programs.

Sprite
A mobile object block that is defined using **POKE**.

SQR
Produces the square root of the number that follows it.

Statement
An instruction in a program. There may be more than one statement in each program line.

STEP
Sets the step size in a **FOR ... NEXT** loop.

STOP
Halts a program and **PRINT**s out the line number in which it appears.

String
A sequence of characters treated as a single item – someone's name, for instance.

Subroutine
A part of a program that can be called when necessary, to produce a particular display or carry out a number of calculations repeatedly, for example.

Syntax
Rules governing the way statements must be put together in a computer language.

TAB
Positions text along a line.

Variable
A labeled slot in the computer's memory in which information can be stored and retrieved later in a program.

VERIFY
Checks that a program has been recorded correctly on a tape or disk using **SAVE**.

VIC
Video Interface Circuit; the chip responsible for controlling sprites.

INDEX

Acknowledgments

Dorling Kindersley would specially like to thank Ian Graham for his significant contribution to this series. Thanks are also due to Philip Freebrey and Paul Rubert for technical asssistance, to Fred Gill for checking the text, and to Richard Bird for preparing the index. Commodore Business Machines (UK) Ltd kindly helped in the supply of equipment.

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

COMMODORE 64

THE DK SCREEN-SHOT PROGRAMMING SERIES

Never has there been a more urgent need for a series of well-produced, straightforward, practical guides to learning to use a computer. It is in response to this demand that The DK Screen-Shot Programming Series has been created. It is a completely new concept in the field of teach-yourself computing. And it is the first comprehensive library of highly illustrated, machine-specific, step-by-step programming manuals.

BOOKS ABOUT THE COMMODORE 64

This is Book One in a series of unique step-by-step guides to programming the Commodore 64. Together with its companion volumes, it will build up into a self-contained teaching course that begins with the basic principles of programming, and progresses – via more sophisticated techniques and routines – to an advanced level.

ALSO AVAILABLE IN THIS SERIES

Step-by-Step Programming for the **ZX Spectrum**

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Acorn Electron**

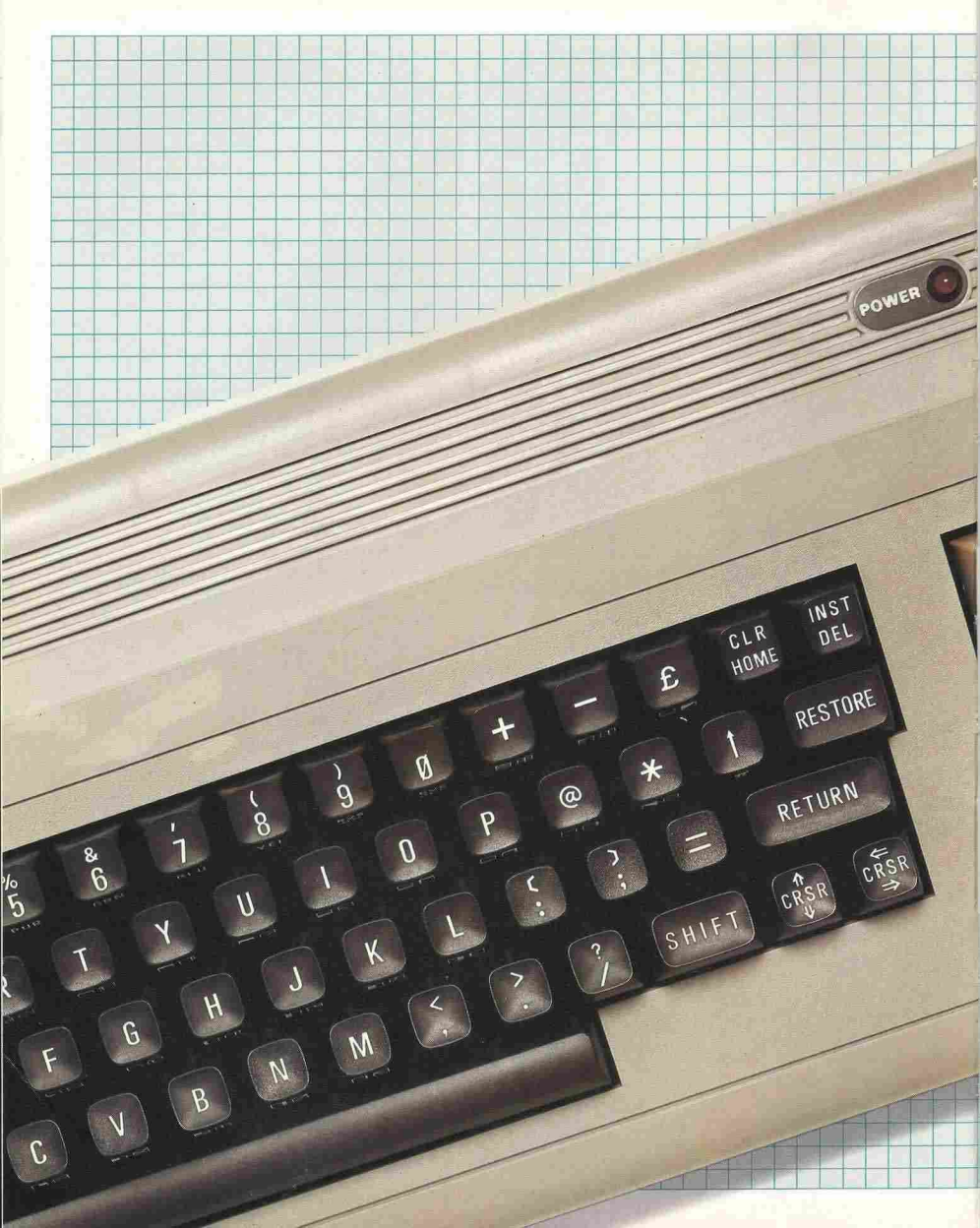
Step-by-Step Programming for the **Apple IIe**

Step-by-Step Programming for the **IBM PCjr**

PHIL CORNES

After taking a B.A. in Mathematics and Computing, Phil Cornes has been involved in system development of computer-based education at British Telecom's National Training College. He has been a part-time technical author since 1978, and has become a regular contributor to personal computer magazines such as *Personal Computer World, Computing Today* and *Electronics Today International*. He has written a book and a large number of articles on programming and using the Commodore 64.

BOOK TWO


POWER
CLR HOME
INST DEL
RESTORE
RETURN
SHIFT
CRSR
CRSR
5
6
7
8
9
T
Y
U
I
O
P
F
G
H
J
K
L
V
B
N
M
C

DK

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

COMMODORE 64

PHIL CORNES

GUILD PUBLISHING LONDON

BOOK TWO

CONTENTS

The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Designer Roger Priddy
Photography Vincent Oliver
Series Editor David Burnie
Series Art Editor Peter Luff
Managing Editor Alan Buckingham

First published in Great Britain in 1984 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

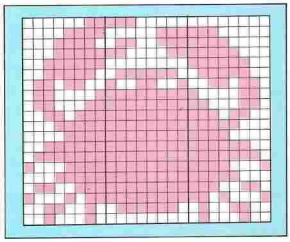
This edition published 1984 by Book Club Associates by arrangement with Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

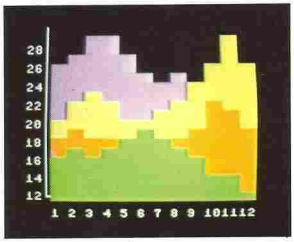
British Library Cataloguing in Publication Data

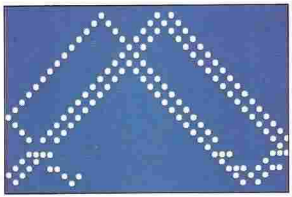
Cornes, Phil
Step-by-step programming for the Commodore 64. Bk. 2
1. Commodore 64 (Computer)—— Programming
I. Title
001.64'2 QA76.8.C64

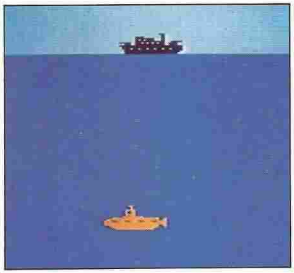
ISBN 0-86318-041-8

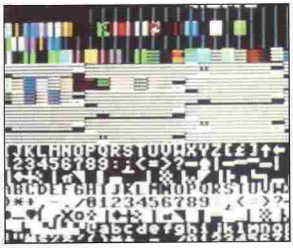
Typesetting by The Letter Box Company (Woking) Limited, Woking, Surrey, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona











DEFINING AND USING FUNCTIONS

All computers feature a range of built-in functions, commands that can be used to transform one number into another in a specific way. Functions produce a result that can be used later in a program. SQR (SQuare Root) and INT (INTeger) are two examples of functions that are pre-programmed on the Commodore. When you use these commands, they take a number and operate on it to produce another number.

The range of built-in functions on the Commodore is quite wide, but if you want to use a function that does not appear in the Commodore's BASIC, you don't have to type out the instructions every time. The Commodore allows you to program it to carry out specific sequences of calculations. These sequences or functions are "called" by the command FN (FunctioN) and are defined by the command DEF FN (DEFine FunctioN).

How to write functions for your programs

To use a function, you must first define what it is going to do. That is done with a defining statement. For instance:

```
120 DEF FNA(X)=4*X+36
```

defines a function called "A". The number that a function operates on is known as its argument. In this case the argument of the function is X. The function takes whatever value of X it is given, multiplies it by 4, and then adds 36. If in a program you wanted to put the number 10 into this function, you would do so by using the keyword FN like this:

```
200 PRINT FNA(10)
```

This would PRINT the value of the function when 10 is substituted for X, which is 4*10+36, or 76.

Once a function has been defined in a program, you can use it and its argument just like any other number or numeric variable. For example, you can add, subtract, divide and multiply functions and their arguments together, and even make functions work on numbers that are themselves functions. Unless you are doing mathematical research you are unlikely to get this far, but for more straightforward tasks functions are easy to use and helpful in making programs simpler.

What can functions do?

The following program shows a simple way in which you can put functions to work to produce a numerical result which is then PRINTed. It takes the distance of a star measured in light years and then converts it into a distance measured in miles. The function that actually does the conversion is defined in line 50. It multiplies the number it is given by 5.88:

STAR DISTANCE PROGRAM

```
LIST
10 PRINT CHR$(147) : POKE 53280,0 : POKE
 53281,0
20 POKE 214,5 : PRINT
30 PRINT TAB(8);"STAR DISTANCE PROGRAM"
40 PRINT TAB(8);"
50 DEF FNC(L)=L*5.88
60 PRINT : PRINT : PRINT
70 PRINT "ENTER STAR'S DISTANCE IN LIGHT
 YEARS":PRINT:PRINT TAB(15);
80 INPUT L
90 PRINT : PRINT : PRINT
100 PRINT "THE STAR IS ";FNC(L);" THOUSA
ND BILLION"
110 PRINT:PRINT TAB(10);"MILES FROM EART
H"
READY.
```

```
STAR DISTANCE PROGRAM

ENTER STAR'S DISTANCE IN LIGHT YEARS
? 34.7

THE STAR IS  204.036  THOUSAND BILLION
MILES FROM EARTH
```

Going to the trouble of using a function here might seem a bit unnecessary, and in fact it's unlikely that you would use FN in such a simple program. But imagine what would happen if you wanted to do the calculation a number of times at different places in the same program, and with different numbers. It is then that the user-defined function really comes into its own. When the function is long and complicated, defining it just once enables you to make calculation lines much simpler to write and check. FN is very much like a one-command subroutine that deals only with numbers.

Because an expression containing FN actually represents a number, you can use it to replace any kind of complex calculation. When you write your own functions, you are in effect giving the computer functions that its resident programming language, BASIC, doesn't already have – extending the capabilities of the language.

Using functions in a calculation sequence

Imagine that you want to calculate the cost of something that is sold by area – perhaps carpets to cover the floors of a house. You would need to multiply the length and width of each room to get its area, and then multiply that by the cost of the floor covering per unit area. If you called the length and the width X and Y, and the cost per unit area Z, then the cost per room would be worked out by:

(X*Y)*Z

In the next program, the cost for each room is calculated by a function. It is defined in line 10 as function C. This function is used right at the end of the program in lines 340 and 350, after values for X and Y have been supplied, together with the value for cost per unit area, P, which is asked for by line 50:

CARPET COSTER PROGRAM

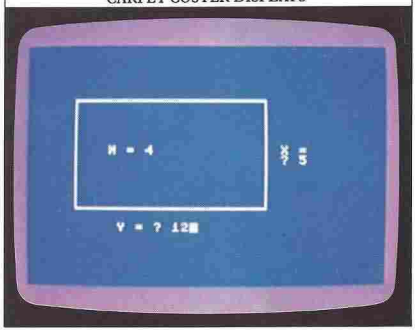
```
10 PRINT CHR$(147) : DEF FNC(Z)=(X*Y)*Z
: POKE 53280,0 : POKE 53281,0
20 PRINT "CARPET COSTING PROGRAM" : PRIN
T : PRINT
30 INPUT "HOW MANY ROOMS TO CARPET";N :
PRINT
40 PRINT "WHAT IS THE COST OF CARPET"
50 INPUT "PER UNIT AREA";P
60 T=0 : FOR C=1 TO N
70 PRINT CHR$(147)
80 POKE 214,4 : PRINT : POKE 211,5 : POK
E 53280,4 : POKE 53281,6
90 PRINT CHR$(176);
100 FOR K=1 TO 20
110 PRINT CHR$(99); : NEXT K
120 PRINT CHR$(174)
130 FOR K=1 TO 10
140 POKE 211,0 : PRINT CHR$(98);
150 POKE 211,26 : PRINT CHR$(98)
160 NEXT
170 POKE 211,5 : PRINT CHR$(173);
180 FOR K=1 TO 20

READY.
```

```
LIST 190-

190 PRINT CHR$(99); : NEXT K
200 PRINT CHR$(189)
210 POKE 214,9 : PRINT : POKE 211,9
220 PRINT "W = ";
230 POKE 214,2 : PRINT : POKE 211,6
240 POKE 214,5 : PRINT : POKE 211,28
250 PRINT "W = ";
260 POKE 211,28 : INPUT X
270 POKE 214,17 : PRINT : POKE 211,10
280 INPUT "W = ";Y
290 PRINT CHR$(147) : POKE 53280,0 : POK
E 53281,0
300 POKE 214,8 : PRINT : POKE 211,8
310 PRINT "COST PER UNIT AREA";P
320 PRINT : PRINT TAB(8);"LENGTH";X
330 PRINT : PRINT TAB(8);"WIDTH";Y
340 PRINT : PRINT TAB(8);"COST";FNC(P)
350 T=T+FNC(P) : PRINT : PRINT
360 PRINT TAB(8);"TOTAL COST";T
370 FOR Z=1 TO 2000 : NEXT Z : NEXT C
READY.
```

Lines 70 to 290 set up a graphics display which produces the outline of a room, and then wait for you to key in values for width and length (you can use any units you like, as long as you also use them for the unit area price). Once you have entered the two figures, the program INPUTs the values and then clears the screen.

The next display takes the cost per unit area – square yard, square meter or whatever you have used – and then uses it to tell you what it would cost to cover the room. As well as using the function C to produce this, it uses it again in line 350 to update the running total. The program passes around this loop once for each room in turn. After each pass you will find that the TOTAL COST line in the second display will be updated to show the running total of all the costs calculated:

CARPET COSTER DISPLAYS



You can define a function at any point in a program, although the line containing the function definition must be carried out before the function can be used. This means that it is usually best to put your function definitions near to the beginning of your programs. In the carpet coster program, using a function makes lines 340 and 350 simpler; with really complex calculations functions are essential for making programs easy to understand.

EXTENDING BASIC DECISIONS

The BASIC keywords IF and THEN let a program operate in one way until the condition specified by the IF statement is encountered. When this happens, the program is then triggered to follow another course of action. But the capabilities of IF ... THEN do not stop at making a straightforward "yes" or "no" decision. By combining IF ... THEN with the keywords AND and OR you can make the commands tackle much more complicated situations.

Because BASIC is designed to reflect how words are used in ordinary language, you can use IF ... THEN just as you would when describing a set of conditions to someone. Here is a program which shows how you can take IF ... THEN decision-making to a more advanced level:

BOUNCING BALL PROGRAM

```
LIST -200

10 S=54272 : POKE S+24,15
20 POKE S+5,0 : POKE S+6,240
30 POKE 53280,0 : POKE 53281,0
40 POKE S+4,16 : PRINT CHR$(147)
50 FOR R=3 TO 18
60 POKE 214,R : PRINT CHR$(129)
70 PRINT TAB(11);CHR$(18);" ";
80 PRINT TAB(26);" "
90 POKE 214,3 : PRINT : POKE 211,R+8
100 PRINT CHR$(18);" "
110 POKE 214,18 : PRINT : POKE 211,R+8
120 PRINT CHR$(18);" "
130 NEXT R
140 X1=INT(RND(0)*10)+14 : Y1=15
150 D1=0.9 : D2=-1
160 X2=14 : Y2=INT(RND(1)*10)+6
170 D3=-1 : D4=0.8
180 POKE 214,Y1 : PRINT : POKE 211,X1
190 PRINT "A" : POKE S+4,16
200 POKE 214,Y2 : PRINT : POKE 211,X2

READY.
```

```
LIST 210-

210 POKE 214,Y : PRINT : POKE 211,X
220 PRINT CHR$(113)
230 IF X<13 OR X>25 THEN DX=-DX : F=1
240 IF Y<5 OR Y>16 THEN DY=-DY : F=1
250 IF F=0 THEN 180
260 POKE S+4,17
270 GOTO 180
READY.
```

Lines 40 to 140 simply set up the display – an orange outline box in the middle of the screen. Lines 160 and 170 specify the starting position and direction for a ball (a keyboard graphics character) inside the box. To make the ball appear to move, line 200 continually produces new co-ordinates.

After this happens, lines 230 and 240 check whether the ball has reached any of the box's walls. They examine the ball's position to see if the row number is one below the box lid or one above the box bottom, or if its column number is one more than the left side or one less than the right side. If any of these conditions is met, lines 230 and 240 reverse the ball's vertical or horizontal motion, whichever is necessary.

How to add decisions together

You can now move on a stage further from the previous program to see how more conditions can be incorporated in an IF ... THEN statement. The next program uses AND to test whether or not a series of statements are true all at the same time:

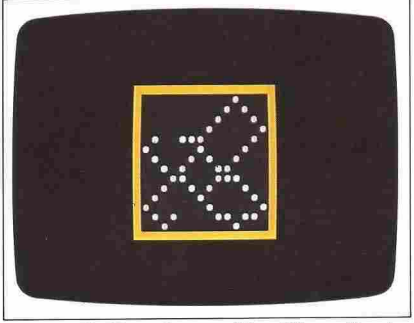
DOUBLE BOUNCING BALL PROGRAM

```
LIST -200

10 S=54272 : POKE S+24,15
20 POKE S+1,115 : POKE S,88
30 POKE S+5,0 : POKE S+6,240
40 POKE 53280,0 : POKE 53281,0
50 PRINT CHR$(147)
60 FOR R=3 TO 18
70 POKE 214,R : PRINT CHR$(129)
80 PRINT TAB(11);CHR$(18);" ";
90 PRINT TAB(26);" "
100 POKE 214,3 : PRINT : POKE 211,R+8
110 PRINT CHR$(18);" "
120 POKE 214,18 : PRINT : POKE 211,R+8
130 PRINT CHR$(18);" "
140 NEXT R
150 PRINT CHR$(159)
160 X=15 : Y=15
170 DX=1.2 : DY=-1
180 POKE 214,Y : PRINT : POKE 211,X
190 POKE S+4,16
200 X=X+DX : Y=Y+DY : F=0

READY.
```

```
LIST 210-

210 PRINT " "
220 X1=X1+D1 : Y1=Y1+D2 : F=0
230 X2=X2+D3 : Y2=Y2+D4
240 POKE 214,Y1 : PRINT : POKE 211,X1
250 PRINT CHR$(156);CHR$(113)
260 POKE 214,Y2 : PRINT : POKE 211,X2
270 PRINT CHR$(159);CHR$(113)
280 IF X1<13 OR X1>25 THEN D1=-D1 : F=1
290 IF Y1<5 OR Y1>16 THEN D2=-D2 : F=1
300 IF X2<13 OR X2>24 THEN D3=-D3 : F=2
310 IF Y2<5 OR Y2>17 THEN D4=-D4 : F=2
320 IF X2<=X1+1 AND X2>=X1-1 AND Y2<=Y1+
1 AND Y2>=Y1-1 THEN F=3
330 IF F=0 THEN 180
340 IF F=1 THEN POKE S+1,115 : POKE S,88
  : GOTO 370
350 IF F=2 THEN POKE S+1,55 : POKE S,80
  : GOTO 370
360 POKE S+1,85 : POKE S,88
370 POKE S+4,17 : IF F=3 THEN 10
380 GOTO 180
READY.
```

This program is very similar to the first one, except that now there are two sets of lines that PRINT a graphics ball at changing row and column numbers. And in this program, each of the balls starts at a random co-ordinate which is defined in lines 140 and 160. The balls are then animated by lines 180 to 380. This display shows what happens if you delete the erasing statements in lines 190 and 210:

DOUBLE BOUNCING BALL DISPLAY



The second ball is made to set off in a different direction from the first and at a slightly different vertical speed, so that the two balls have a greater chance of meeting. Otherwise, they would just follow each other around the box in the same tracks. Line 320 is the one in which the computer makes a multiple decision about the position of both balls. Without this line, when the balls met, they would just carry on through each other as if nothing had happened. This isn't a very convincing simulation of what would really occur, so line 320 decides whether the balls are close enough together to have collided. The line includes an IF...THEN decision with three ANDs to see if X1 and X2 are sufficiently close together, and then if Y1 and Y2 are also within the same limits. It does this by taking X2, for example, and then deciding whether it is smaller or equal to X1+1 and simultaneously greater or equal to X1−1.

If all these conditions are met, then it means that the two balls are either occupying the same position or are at adjacent positions, in which case they can be assumed to have collided. A bleep is sounded and then the whole process starts again.

IF...THEN in games programming

So, as you've just seen, IF...THEN is very useful if you want to know whether or not two characters are occupying the same screen location. This is often used in programs in which one character is "shot down" by another:

FIRE-BASE PROGRAM

```
10 PRINT CHR$(147);CHR$(158)
20 POKE 53280,5 : POKE 53281,0
30 POKE 214,20 : PRINT
40 PRINT TAB(18);CHR$(191);CHR$(18);CHR$
(183);CHR$(191)
50 PRINT TAB(17);CHR$(191);CHR$(18);CHR$
(184);" ";CHR$(184);CHR$(191)
60 X=7 : Y=19
70 POKE 214,Y : PRINT
80 PRINT TAB(19);" "
90 POKE 214,6 : PRINT
100 PRINT TAB(X);" "
110 X=X+1 : Y=Y-1
120 IF X>26 THEN X=5
130 IF Y<1 THEN Y=19
140 POKE 214,6 : PRINT
150 PRINT TAB(X);CHR$(153);"I>"
160 POKE 214,Y : PRINT
170 PRINT TAB(19);CHR$(5);"↑"
180 IF X=18 AND Y=6 OR X=19 AND Y=6 THEN
200
190 GOTO 70
200 FOR T=1 TO 1000 : NEXT T : GOTO 10
READY.
```

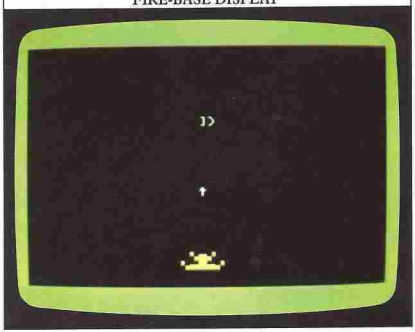
This program PRINTs a fire-base at the bottom of the screen. It fires upward arrows at a horizontal target that repeatedly flies across the screen. Line 180 checks whether the screen co-ordinates of the upward arrow are the same as those of the target. If they are, the program jumps right back to line 10 and begins again. If not, it jumps back to line 70 and moves all the characters on one space. Line 130 checks whether the upward arrow has reached the top of the screen, and line 120 checks whether the target has reached the right edge.

When you RUN the program, you should find that the fire-base's arrow scores a direct hit on the horizontal target's fourth pass across the screen. This happens because the program is working with fixed figures. If instead you use the following line:

```
60 X=INT(RND(0)*10+1):Y=19
```

the results become unpredictable and will change with each RUN:

FIRE-BASE DISPLAY



SCANNING THE KEYBOARD

To key in new information while a program is RUNning you have – until now – used the command INPUT. With INPUT you must press RETURN after keying the information in. This technique has its disadvantages. Even when you know that it's necessary to use RETURN, you can occasionally forget, and using two keys in sequence also slows programs down. It is much more useful if you make the Commodore respond without waiting for you to press RETURN, just as arcade machines respond every time you press a button. To make the Commodore do this, you can use the keyword GET.

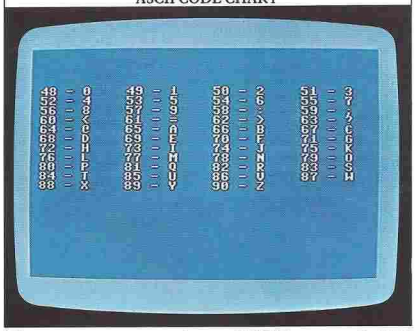
How the computer recognizes characters

As you may have found out from Book 1, all the symbols that the Commodore recognizes are stored as code numbers from 0 to 255, according to the ASCII (American Standard Code for Information Interchange) system. The computer uses this coding system every time you INPUT a number or string, to determine its content. There's a complete ASCII chart on page 61, but you can quickly get the computer to PRINT the numbers and letters with their codes by keying this in:

```
10 FOR N=48 TO 90
20 PRINT N;" - ";CHR$(N),
30 NEXT N
```

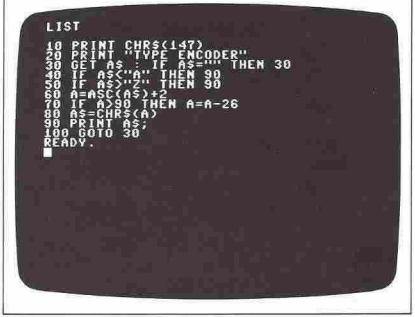
A partial ASCII chart then appears on the screen:

ASCII CODE CHART



The next program uses the two BASIC commands GET and ASC to respond when you press character keys by taking the character's code, changing it and then PRINTing the character specified by the new code. The result is a keyboard encoder which produces a coded message as you type:

KEYBOARD ENCODER PROGRAM



In this program line 20 PRINTs a heading, and then line 30 uses GET to scan the keyboard. Any character that is entered by a key-press when the scan takes place is labeled with the variable A$. If you don't press a key during this time, then the variable is given a zero value. The second part of line 30 tests to see if the variable has a zero value, and if it has, the program loops back again so that in effect the keyboard is continually scanned.

When the computer detects a key-press, program control is passed to line 40. If the key pressed is not a letter, lines 40 and 50 pass control to line 90 which just PRINTs the character detected. If the key-press is a letter, line 60 converts this character to its ASCII number using the ASC command (this works like CHR$ in reverse). It then makes this the value of the variable A. Lines 70 and 80 then operate on A to encode it into a new letter, and then convert it back to a character, A$, which is PRINTed by line 90.

This program seems to PRINT nonsense when you type a message into it. There's not much point in a code that can't be decoded, and you can quickly turn your program into a decoder by changing the following three lines:

```
20 PRINT "TYPE DECODER"
60 A=ASC(A$)-2
70 IF A<65 THEN A=A+26
```

To see this at work, try keying in the following when the decoder program is RUNning:

```
VJKU KU C VGUV OGUUCIG
```

How to make the function keys work

As well as using GET to scan for the letter and number keys, you can also use it to find out if any of the function

keys have been pressed, because each of these keys has an ASCII value although normally they do not PRINT anything on the screen. The ASCII values of the function keys range from 133 to 140, as this program will demonstrate when you RUN it:

FUNCTION KEY DETECTION PROGRAM

```
LIST

10 PRINT CHR$(147)
20 GET A$ : IF A$="" THEN 20
30 A=ASC(A$)
40 IF A=133 THEN PRINT "F1" : GOTO 20
50 IF A=137 THEN PRINT "F2" : GOTO 20
60 IF A=134 THEN PRINT "F3" : GOTO 20
70 IF A=138 THEN PRINT "F4" : GOTO 20
80 IF A=135 THEN PRINT "F5" : GOTO 20
90 IF A=139 THEN PRINT "F6" : GOTO 20
100 IF A=136 THEN PRINT "F7" : GOTO 20
110 IF A=140 THEN PRINT "F8" : GOTO 20
120 PRINT "NOT A FUNCTION" : GOTO 20
READY.
```

Every time you press one of the function keys, its identity appears on the screen. This may not seem very useful, but it actually shows you how these keys can be used in programming. If you use GET you can make the function keys control programmed operations. You can have a total of eight separate functions from, for example, color setting to sprite animation.

Testing your reactions with GET

The Commodore's "jiffy" clock, detailed in Book 1, is a three-byte counter that is incremented 60 times per second to keep a record of the time elapsed since the computer is switched on. Also built into the Commodore are some simple facilities that enable you to access the clock and use the numbers that it stores. Try this direct command:

PRINT TI

What you see when you do this is a number that tells you how many sixtieths of a second have elapsed since you started up the computer. Now type in another line:

PRINT TI$

This time what is PRINTed is the same information but converted into hours, minutes and seconds, with two digits for each. So 013000, for example, means that it is exactly 1½ hours since you started using the computer. As well as being able to read the clock in these two ways, you can also reset it to zero, or indeed any other number you want, by this kind of command:

TI$="000000"

This sets the clock to zero.

You can use GET in combination with the clock to time the speed of anything during a program. One of the simplest ways of doing this is in a reaction test program. Here's one that produces a random letter and then times how long it takes you to find and press the key that it has selected:

REACTION TESTER PROGRAM

```
LIST

10 PRINT CHR$(147) : POKE 53280,0 : POKE
 53281,0
20 POKE 214,5 : PRINT : POKE 211,6
30 PRINT "TEST YOUR REFLEXES AGAINST"
40 PRINT : PRINT TAB(10);
50 PRINT "THE REACTION TESTER"
60 FOR T=1 TO 1500 : NEXT T
70 A$=CHR$(INT(RND(0)*26)+65)
80 POKE 214,10 : PRINT : POKE 211,10
90 PRINT "FIND THIS KEY:  ";
100 FOR T=1 TO 500 : NEXT T
110 PRINT A$ : TI$="000000"
120 GET R$ : IF R$="" THEN 120
130 IF R$<>A$ THEN 120
140 POKE 214,14 : PRINT : POKE 211,8
150 PRINT "YOU TOOK";INT(TI/0.6)/100;
160 PRINT "SECONDS"
170 FOR T=1 TO 3000 : NEXT T
180 GOTO 10
READY.
```

```
TEST YOUR REFLEXES AGAINST
    THE REACTION TESTER

FIND THIS KEY:  I

YOU TOOK .94 SECONDS
```

Lines 10 to 60 clear the screen, PRINT the program title on the screen, give a two-second pause and then the game begins. Line 70 generates a random letter. This is done by CHR$ which converts a randomly generated number from 65 to 90 into a single character that becomes A$. Lines 90 to 110 PRINT a message asking you to locate the random letter and set the clock to zero. When you press the right key, the value of the clock is read and PRINTed by line 150. The calculation:

INT(TI/0.6)/100

is used to convert the time to seconds and cut the answer to two figures after the decimal point. If you can get a time score below 0.5 seconds, you're probably a touch-typist!

BIT MASKING

On page 8, you encountered two new keywords, AND and OR, and you saw there how they can be used in decision-making. But there is more to AND and OR than this. They are actually both examples of "logical operators", keywords that combine a pair of numbers in special ways. To understand how the Commodore is programmed to produce high-resolution graphics involving plotting points or drawing lines, you will need to know how to use AND and OR to transform numbers into patterns on the screen.

When you PEEK a value from a register, you can simply treat its byte as a single number, disregarding the fact that it is actually 8 separate binary bits. Similarly, you can POKE a whole byte into a register so that all the bits that the register previously held are lost from memory. These techniques are fine for some applications, but fairly soon you will find that you want to set or reset a single bit within the memory without changing all the other bits in its byte. With the Commodore, this is a crucial part of advanced programming. To do it, you need to use AND and OR in a technique called "bit masking".

How to make a Commodore alter a single bit

You may remember from Book 1 that V+21 (where V is the VIC chip base address of 53248) is the register that controls which of 8 sprites are switched on and which are switched off.

SPRITE CONTROL BITS

Each sprite is turned on or off by a single bit in register V+21 of the VIC chip.

Bit number	Sprite controlled by bit	Decimal value of bit
0	0	1
1	1	2
2	2	4
3	3	8
4	4	16
5	5	32
6	6	64
7	7	128

In order to be able to control the individual sprites independently, you need to control the individual bits of the V+21 register, turning them on or off in a program.

This is where the AND and OR operators come in. These two operators work by comparing two numbers bit by bit and producing results based on the comparisons. The AND operator puts a 1 bit in the result in every bit position where there is a 1 bit in *both* of the numbers being compared. The OR operator puts a 1 bit in the result in every position where there is a 1 bit in *either* of the two numbers being compared.

To take an example, if you want to set bit 4 in a byte to a value of 1 without affecting the rest of the byte, you could make the computer do it like this:

```
200 BYTE=BYTE OR 16
```

The decimal value of bit 4 is 16 (2 ↑ 4). If the byte has a value of say, 164, the line will work like this:

HOW OR WORKS

The OR function gives a result bit of 1 when either or both bits it is working on have a value of 1.

Bit number	7	6	5	4	3	2	1	0	
BYTE=164=	1	0	1	0	0	1	0	0	
16=	0	0	0	1	0	0	0	0	OR
RESULT=	1	0	1	1	0	1	0	0	

Bit 4, which was originally 0 or "off", is now 1 or "on", while the rest of the byte is unchanged.

Converting the numbers to binary makes the effect of the OR function quite easy to follow. If you convert it back to decimal, it gives 164 OR 16=180, a result that you can check by getting the computer to PRINT it out.

In general, to set or "turn on" any specific bit(s) in a byte you need to OR the byte with the number that has the required bit or bits set. Conversely, to reset or "turn off" given bit(s) in a byte, you need to AND the byte with 255 minus the number that has the required bit or bits set. So, taking the previous result of 180, if you wanted to reset bit 5 you would need to AND 180 with the value 255−32=223:

HOW AND WORKS

The AND function gives a result bit of 1 only when both bits it is working on have a value of 1.

Bit number	7	6	5	4	3	2	1	0	
BYTE=180=	1	0	1	1	0	1	0	0	
255−32=223=	1	1	0	1	1	1	1	1	AND
RESULT=	1	0	0	1	0	1	0	0	

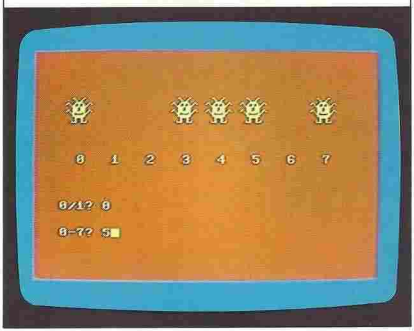
Turning sprites on and off

So much for the theory. To see how AND and OR can be used in a program, try out the following listing. It uses VIC chip location V+21 together with INPUTs from the keyboard to turn 8 sprites on and off instantaneously as you choose:

USING **AND** AND **OR** WITH SPRITES

```
LIST -180

10 PRINT CHR$(147); : POKE 53280,6
20 POKE 53281,2 : FOR C=0 TO 62
30 READ BYTE
40 POKE 832+C,BYTE
50 NEXT C : V=53248 : POKE V+16,0
60 FOR C=0 TO 7
70 POKE 2040+C,13
80 POKE V+39+C,7
90 POKE V+2*C+1,90
100 HX=INT((32*C+54)/256)
110 POKE V+2*C,(32*C+54)-(HX*256)
120 POKE V+16,PEEK(V+16) OR (2↑C*HX)
130 POKE V+21,PEEK(V+21) OR 2↑C
140 NEXT C : PRINT CHR$(158);
150 POKE 214,10 : PRINT : POKE 211,1
160 FOR C=0 TO 7
170 PRINT TAB(C*4+4);C;
180 NEXT C : PRINT
READY.
```

```
LIST 190-

190 POKE 214,15 : PRINT : POKE 211,3
200 PRINT "0/1 ";
210 POKE 211,6 : INPUT A
220 POKE 214,18 : PRINT : POKE 211,3
230 PRINT "0-7";: POKE 211,6 : INPUT S
240 POKE 214,18 : PRINT : POKE 211,3
250 PRINT "
    "
260 IF A=1 THEN 290
270 POKE V+21,PEEK(V+21) AND (255-2↑S)
280 GOTO 190
290 POKE V+21,PEEK(V+21) OR 2↑S
300 GOTO 190
800 DATA 3,0,192,0,129,0,0,66,0
810 DATA 0,126,0,36,255,6,145,153,137
820 DATA 11,153,208,7,255,224,3,231,192
830 DATA 3,231,192,99,231,198,147,255,20
1
840 DATA 11,126,208,7,0,224,3,255,192
850 DATA 3,255,192,0,255,0,0,195,0
860 DATA 0,195,0,4,195,32,7,195,224
READY.
```



Seeing computer logic at work

The next program allows you to see the calculations that bit masking carries out, this time PRINTed on the screen. When you RUN it, you are asked for three INPUTs. The first is the choice of AND or OR operator, and the other two are the numbers that the chosen operator is to operate on. The program then converts your decimal numbers into binary, performs the operation and converts the result back into decimal:

AND/OR CALCULATOR PROGRAM

```
10 PRINT CHR$(147);CHR$(5)
20 POKE 214,5 : PRINT : POKE 211,2
30 INPUT "DO YOU WISH TO SEE AND/OR (A/O
)";A$
40 INPUT "  ENTER 1ST VALUE (0-255)";V1
50 INPUT "  ENTER 2ND VALUE (0-255)";V2
60 PRINT : PRINT TAB(4);V1;TAB(9);
70 V=V1 : GOSUB 500 : PRINT
80 PRINT TAB(4);V2;TAB(9);"+"
90 V=V2 : GOSUB 500 : PRINT
100 IF A$="A" THEN 130
110 PRINT TAB(3);"=";V1 OR V2;TAB(9);
120 V=V1 OR V2 : GOSUB 500 : GOTO 150
130 PRINT TAB(3);"=";V1 AND V2;TAB(9);
140 V=V1 AND V2 : GOSUB 500
150 PRINT : PRINT : PRINT TAB(2);
160 INPUT "PRESS RETURN TO CONTINUE";A$
170 GOTO 10
500 FOR C=7 TO 0 STEP -1
510 PRINT SGN(V AND 2↑C);
520 NEXT C : PRINT : RETURN
READY.
```

This program uses a function at line 510 which will be new to you. This is SGN, and it is in the line that converts from decimal into binary. The decimal number is contained in variable V and the binary bit number to be PRINTed is in the variable C. The expression:

V AND (2 ↑ C)

takes the Cth bit from the value of V. If the Cth bit is set to 1 then 2 ↑ C is returned. If the Cth bit is reset to 0 then the value 0 is returned. 2 ↑ C and 0 then need to be translated into 1 and 0 for PRINTing. This is where SGN comes in. It returns the value 1 if its argument is positive, −1 if it is negative and 0 if it is 0. In this program it therefore only returns two answers, 0 and 1:

AND/OR CALCULATOR DISPLAY

```
DO YOU WISH TO SEE AND/OR (A/O)? O
ENTER 1ST VALUE (0-255)? 219
ENTER 2ND VALUE (0-255)? 197

   219  1  1  0  1  1  0  1  1
   197  1  1  0  0  0  1  0  1
 = 223  1  1  0  1  1  1  1  1

PRESS RETURN TO CONTINUE? ■
```

HIGH RESOLUTION

In all the graphics you have seen so far, the displays have used only predefined characters in low (text) resolution, or sprites. But as well as these two graphics modes, the Commodore also has a very powerful high-resolution graphics facility. Now you have seen how bit masking works you will be able to set up the high-resolution screen; in the following pages you will see how to use it.

Screen layout

The Commodore's high-resolution graphics screen is based on a rectangular grid of 64000 individually controllable pixels, or picture elements. These are laid out as 200 rows with 320 pixels on each row. (A complete high-resolution screen grid is shown on page 58.) Each pixel on the screen requires a single bit in memory to control it. If the bit is set to 1, then the pixel is lit, while if it is reset to 0, the pixel goes blank. At 1 bit each, the 64000 pixels require 64000/8 or 8000 bytes of memory. These 8000 bytes of memory are controlled by the VIC chip, the same chip that controls sprites.

To produce high-resolution graphics, you need to take over an 8000-byte chunk of memory. Unfortunately, the best area of memory for this is already used for storing your BASIC programs. However, they don't have to be stored here, and you can easily make the BASIC ROM store them somewhere else.

How to move the BASIC storage area

Rearranging program storage is quite simple, as the following screen shows. (If you do have any problems at all with high resolution, just press RUN/STOP and RESTORE together. This will return you to low resolution so you can check your programs for bugs.)

MOVING PROGRAM STORAGE

```
POKE 642,64
READY.
POKE 44,64
READY.
POKE 16384,0
READY.
NEW
READY.
```

It is *very important* to remember that before you start working on programs with high-resolution graphics you must type in this series of commands. If you don't, your programs may well "crash", because the computer will try to store your programs and high-resolution graphics in the same place.

All the rest of the programs in this book are written assuming that you have already keyed in these commands.

One further point to note here is that this sequence must be typed in as direct commands, not as part of a program. Because the commands move several memory pointers that BASIC uses, if you key in the commands as a program, BASIC will not know where the rest of the program is.

Now to continue. All of the facilities for high-resolution graphics that follow in this book will use one or more of a series of subroutines. You will encounter these as you go along; they all have non-overlapping line numbers so that they may all be used as one program. If you store them together on a tape or disk, you will be able to call any of the subroutines as required.

You can add each successive subroutine to the stack you have SAVEd by keying it in, LOADing the stack accumulated so far, and then pressing RETURN with the cursor at each of the new subroutine's lines. This will combine the two programs. Don't LIST the stack of subroutines before you do this or the new one will scroll irretrievably off your screen.

How to clear out the memory

The first pair of subroutines needed to set up the high-resolution screen tells the VIC chip where to find the high-resolution graphics memory, and how to clear the screen by emptying part of the memory:

CLEARING THE MEMORY (SUBROUTINES 1–2)

```
LIST

10 GOSUB 100
20 COL=18 : GOSUB 200
30 END
100 POKE 53272,PEEK(53272) OR 8
110 POKE 53265,PEEK(53265) OR 32
120 RETURN
200 FOR MEM=8192 TO 16191
210 POKE MEM,0 : NEXT MEM
220 FOR MEM=1024 TO 2023
230 POKE MEM,COL : NEXT MEM
240 RETURN

READY.
```

Lines 100 and 110 tell the VIC chip to use the high-resolution display mode and tell it that the 8000 bytes of memory for this start at location 8192. The other subroutine starts at line 200. Lines 200 and 210 clear the screen by POKEing zeros into every bit in the 8000 bytes. Lines 220 and 230 then go through another 1000-byte block of memory from location 1024 to 2023 setting each byte to the value of a variable called COL. This block stores the display codes of the ordinary characters and predefined graphics shapes that appear on the screen. In high-resolution mode this block of memory has a new function. It determines the colors used to draw the pixels. Having 1000 bytes of color memory and 8000 bytes of pixel memory means that each byte of color memory controls the foreground and background color for 8 bytes of pixel memory.

In each byte of color memory bits 4–7 control the foreground color of an 8×8 pixel block, while bits 0–3 control its background color. By POKEing selected numbers into the color memory with the variable COL you can create any foreground and background color combination.

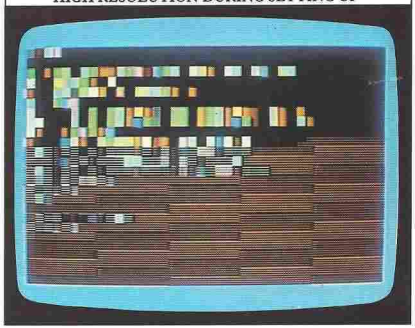
HIGH RESOLUTION COLOUR CODES

To select any of these combinations, two color codes must be added together and POKEd into the screen memory.

Color	Foreground code	Background code
Black	0	0
White	16	1
Red	32	2
Cyan	48	3
Purple	64	4
Green	80	5
Blue	96	6
Yellow	112	7
Orange	128	8
Brown	144	9
Light red	160	10
Dark gray	176	11
Medium gray	192	12
Light green	208	13
Light blue	224	14
Light gray	240	15

If you wanted to draw in white on a red background, the color combination would be 16+2, or 18. This value then needs to be POKEd into all 1000 locations of color memory. This is what lines 220 and 230 of the previous program do. Clearing the memory and setting up color is quite time-consuming, with 9000 separate POKEs to be carried out. The only way you can speed this up is to use machine code, so an alternative to the BASIC subroutine is shown at the end of this page. The next screen shows the process in BASIC; if you use machine code, clearing happens so quickly that you won't see this display:

HIGH RESOLUTION DURING SETTING UP



Now that you have got a program that can move the BASIC storage area, switch the VIC chip to high resolution, clear the screen and set its colors, you are ready to add some further subroutines that will make the computer produce graphics. For safety, you should SAVE the subroutines so far. If you don't do this, don't type NEW before moving on!

Setting up high resolution with machine code

The screen below shows a machine code alternative to the BASIC high-resolution subroutine 2. Its numbering is fully compatible with all the high-resolution subroutines in this book, so you can key it in as a single self-contained unit without having to understand how it works.

The listing here is actually a BASIC program which calls a machine code operation by using the command SYS. The DATA statements are used to POKE numbers into a particular set of locations, and these set up the high-resolution screen. Once the machine code has been carried out, the computer returns to functioning in BASIC as before.

SUBROUTINE 2 (MACHINE CODE)

```
LIST

200 DATA 0,165,252,197,254,208,7,165
210 DATA 251,197,253,208,1,96,160,0
220 DATA 173,80,195,145,251,230,251
230 DATA 208,232,230,252,76,81,195
240 RESTORE : FOR C=50000 TO 50029
250 READ BYTE : POKE C,BYTE : NEXT C
260 POKE 251,0 : POKE 252,4 : POKE 253,2
32
270 POKE 254,7 : POKE 50000,COL : SYS 50
001
280 POKE 251,0 : POKE 252,32 : POKE 253,
64
290 POKE 254,63 : POKE 50000,0 : SYS 500
01 : RETURN

READY.
```

POINT GRAPHICS

On all microcomputers, graphics are produced by lighting up a specified series of pixels. To light an individual pixel with the Commodore, you need to work out which of the 8000 memory locations controls the pixel you want to light, and then which bit within that location's byte needs to be set to 1.

Because the 64000 pixels are arranged in 200 rows of 320 columns, any pixel on the screen can be specified by a row and column number – just as a text position can. The pixels are usually numbered from 0-319 across and from 0-199 down, so the lowest pixel number (8192) is at the top left corner of the screen.

HIGH-RESOLUTION LOCATIONS

The high-resolution screen has a total of 64000 separately controlled points, running from 0 to 319 horizontally and from 0 to 199 vertically. This chart shows the bytes that control just the top left area of the screen (a full grid appears on page 58).

Horizontal co-ordinate

Vertical co-ordinate	0-7	8-15	16-23	24-31	→	312-319
0	8192	8200	8208	8216	→	8504
1	8193	8201	8209	8217	→	8505
2	8194	8202	8210	8218	→	8506
3	8195	8203	8211	8219	→	8507
4	8196	8204	8212	8220	→	8508
5	8197	8205	8213	8221	→	8509
6	8198	8206	8214	8222	→	8510
7	8199	8207	8215	8223	→	8511
8	8512	8520	8528	8536	→	8824
9	8513	8521	8529	8537	→	8825
10	8514	8522	8530	8538	→	8826
↓	↓	↓	↓	↓		
199	15879	15887	15895	15903		16191

If you had to store all the information for the complete screen so that you could look up the required byte, a huge amount of memory would be needed. Fortunately, you can avoid this by using two equations. The first equation tells you which byte the pixel is in, given its co-ordinates, and the second gives you a bit mask value. You can take this value to set a particular bit to 1 by using the bit masking techniques from pages 12–13. The two equations are ready for use in the next set of subroutines. You should key this set into your computer and then LOAD the first set from page 14. You can then merge the two sets by using the RETURN key to re-enter each line in the new set.

You don't have to understand exactly how these subroutines work. They're just a set of calculations for identifying bits and turning them on and off as required. The point plotter lets you plot pixels with program loops instead of individual POKEs:

POINT PLOTTER (SUBROUTINES 3–5)

```
LIST

300 BYTE=8192+INT(LY/8)*320+INT(LX/8)*8+
(LY AND 7)
310 MASK=2↑(7-(LX AND 7))
320 RETURN
400 GOSUB 300
410 POKE BYTE,PEEK(BYTE) OR MASK
420 CMEM=1024+INT(LY/8)*40+INT(LX/8)
430 POKE CMEM,COL
440 RETURN
500 GOSUB 300
520 POKE BYTE,PEEK(BYTE) AND (255-MASK)
530 RETURN

READY.
```

The point plotter contains three separate subroutines. The first one, starting at line 300, calculates the byte and the mask values for the pixel at screen co-ordinates LX,LY. This subroutine is called by the two other subroutines. The first one at line 400 uses these calculations, and the value of the variable COL, to plot or light the pixel at LX,LY in the chosen color. The last subroutine in the program starts at line 500. This unplots or turns off the pixel.

At this point, it will again help if you SAVE these subroutines after adding them to the previous two, as they will be used frequently on the following pages.

Using plotting in graphics

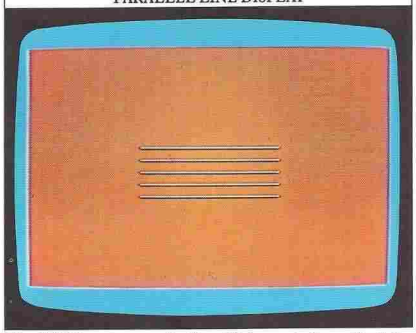
So, assuming that your Commodore now has a total of five separate subroutines in memory, type in and RUN the following program which brings them into action:

DRAWING PARALLEL LINES

```
LIST

10 GOSUB 100
20 COL=18 : GOSUB 200
30 FOR LY=80 TO 120 STEP 10
40 FOR LX=100 TO 220
50 GOSUB 400
60 NEXT LX
70 NEXT LY
80 END
READY.
```

Line 10 sets up the high-resolution screen. Line 20 sets COL to 18 to produce white lines on a red background, and also clears the screen. Line 50 calls the subroutine that plots a pixel and this is contained inside a double set of FOR ... NEXT loops in lines 30 to 70. These two loops generate the values for LX and LY so that the program draws five horizontal lines made up of individual pixels:

PARALLEL LINE DISPLAY



The END statement in line 80 is needed as all of the subroutines follow this program. Without END the program would RUN on into them, disrupting the result.

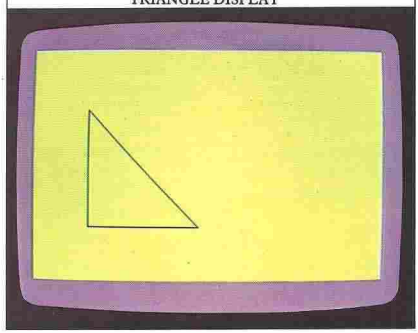
The next program shows how you can link lines up by using these subroutines. This time, lines are plotted vertically and diagonally as well.

As the program is longer than nine lines, it will not fit into the space below the subroutines as the last one did. To overcome this problem, the program is written with higher line numbers than the subroutines, starting at line 1000:

TRIANGLE PROGRAM

```
LIST

10 GOTO 1000
1000 GOSUB 100 : POKE 53280,4
1010 COL=103 : GOSUB 200
1020 LX=50
1030 FOR LY=50 TO 150
1040 GOSUB 400 : NEXT LY
1050 LY=150
1060 FOR LX=50 TO 150
1070 GOSUB 400 : NEXT LX
1080 FOR C=150 TO 50 STEP -1
1090 LX=C : LY=C
1100 GOSUB 400 : NEXT C
1110 GOTO 1110
READY.
```

TRIANGLE DISPLAY



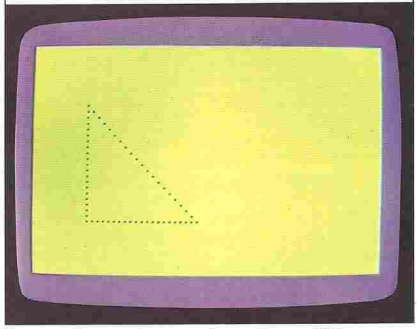
How to change STEP in a loop

You will probably have noticed that the previous programs all draw lines by plotting a series of points in adjacent pixels. This is done by using FOR...NEXT loops. You can modify these programs so that instead of producing solid lines, they produce dotted ones. This is done by setting a STEP size so that the program skips some of the pixels. You can see this in the triangle program if you make the following line changes:

```
1030 FOR LY=50 TO 150 STEP 5
1060 FOR LX=50 TO 150 STEP 5
1080 FOR C=150 TO 50 STEP -5
```

Now you can see that the display is made up of individual pixels, each 5 units apart (you can use this technique to plot stippling inside shapes). Otherwise, the outline is the same:

DISPLAY WITH DOTTED LINES



Because the program now has only one-fifth as many pixels to plot, it RUNs considerably faster.

DRAWING LINES

Now you know how to plot pixels on the screen, you can draw lines and simple shapes by using FOR ... NEXT loops. However, using the techniques decribed on the previous two pages, you can only draw lines vertically, horizontally or diagonally at 45 degrees. To produce graphics, you need a way of drawing lines at any angle. This is what you are going to find out about next – a subroutine that can draw a line between any pair of co-ordinates on the screen.

The drawing subroutine

The basic subroutine for drawing lines appears in the screen below. You should add this to the subroutine program you have on tape or disk, so that you now have six subroutines altogether:

LINE DRAWING (SUBROUTINE 6)

```
LIST

600 GT=ABS(NX-LX)
610 IF ABS(NY-LY)>GT THEN GT=ABS(NY-LY)
620 XINC=(NX-LX)/GT : YINC=(NY-LY)/GT
630 XX=LX+0.5 : YY=LY+0.5
640 FOR CC=1 TO GT
650 LX=INT(XX) : LY=INT(YY) : COSUB 400
660 XX=XX+XINC : YY=YY+YINC
670 NEXT CC : LX=NX : LY=NY : RETURN
READY.
```

Don't worry if you can't understand the lines here. The subroutine is simply a collection of equations that work out where a line of pixels should be plotted. It uses two sets of co-ordinates, LX,LY and NX,NY, and draws a line from the last co-ordinate position to the new one. It also updates the last LX and LY variables, making them equal NX and NY so that another line may be drawn, starting from the position where the previous one finished.

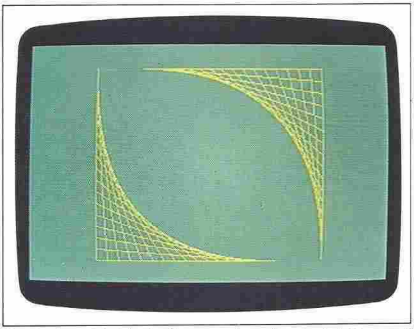
The keyword ABS which appears in lines 600 and 610 gives the absolute value of any number or numeric variable that follows it. In this subroutine it has the effect of making any value of NX–LX or NY–LY positive, so the program can use positive figures to produce lines.

Pin and string patterns

Now that you have all six subroutines available, try adding the following lines, and then RUN the complete program to see the display appear:

PIN AND STRING PROGRAM

```
LIST

10 GOTO 1000
1000 POKE 53280,11 : GOSUB 100
1010 COL=124 : GOSUB 200
1020 FOR C=0 TO 160 STEP 10
1030 LY=20 : NX=260
1040 LX=C+100 : NY=C+20
1050 GOSUB 600
1060 NEXT C
1070 FOR C=0 TO 160 STEP 10
1080 LX=60 : NY=180 : COL=124
1090 LY=C+20 : NX=C+60
1100 GOSUB 600

READY.
```



Producing this kind of pattern is quite straightforward because there is a simple mathematical link between the end points of the various lines. If you vary the numbers used, the display changes. But what if you want to draw something that doesn't fit a simple equation – a space shuttle, for example? For this, you need a subroutine that draws lines that you can specify individually to give any shape you want.

A high-resolution line machine

You can use the same method for storing line information as you would use for storing notes in a sound program – READing a section of DATA. In the next program, the DATA has been written in to produce a design. You can think of this program as producing an imaginary "pen". As with a real pen, it lets you lift it up from the screen, put it back again, or move it in either up or down positions:

DRAWING WITH DATA

```
LIST

10 GOTO 1000
1000 POKE 53280,0 : GOSUB 100
1010 COL=208 : GOSUB 200
1020 PEN=0 : LX=0 : LY=0
1030 READ NX,NY
1040 IF NX>=0 THEN 1090
1050 IF NY=1 THEN PEN=1 : GOTO 1030
1060 IF NY=0 THEN PEN=0 : GOTO 1030
1070 IF NY=2 THEN 1070
1080 COL=-NY : GOTO 1030
1090 IF PEN=0 THEN LX=NX : LY=NY : GOTO
1030
1100 GOSUB 600 : GOTO 1030

READY.
```

```
LIST 1200-

1200 DATA -1,0,49,123,-1,1,74,102,73,112
,78,112,81,98,74,102,88,94,212,78
1210 DATA 207,85,254,4,266,2,256,82,207,
85,-1,0,258,72,-1,1
1220 DATA 272,70,286,98,250,137,236,108,
272,70
1230 DATA 258,72,272,62,304,60,280,86,-1
,0,245,125,-1,1
1240 DATA 90,130,136,156,145,192,173,195
,245,125
1250 DATA -1,0,236,108,-1,1,158,114,-1,0
,105,140,-1,1,78,140,49,134,49,123
1260 DATA -1,0,250,137,-1,1,233,137
1360 DATA -1,2
READY.
```

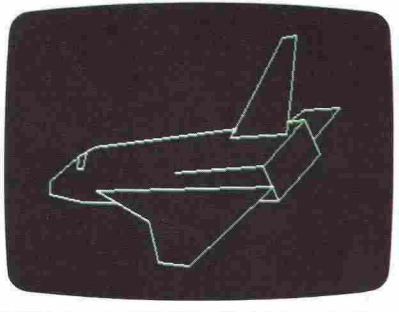
The program also gives you the option of changing the colour of the "ink", and again this is controlled by numbers in the DATA statements, which change the value of the variable COL.

How the line machine works

In this program, all the numbers in the DATA statements are split into pairs. If both the numbers are greater than or equal to zero, then the pen is moved in a straight line to the position specified by the pair of numbers. If the pen is down, then a line will appear.

For all other options, the first of the pair of DATA numbers is −1. The second number then tells the computer what to do. If the second number is 1, the pen is moved down onto the screen. If it is 0, the pen goes up. If the number if negative, the number that follows the minus sign fixes the variable COL, and hence the ink color. Finally, if the second value is 2, the program finishes by going into an endless loop in line 1070. The program terminates like this so that the display will not be spoiled by the READY message appearing on the screen. To stop the program completely, you just need to press the RUN/STOP key. With the DATA lines in the previous screen, the program produces this display:

LINE MACHINE DISPLAY



Remember that to LIST the program to make changes if there are any errors in your version, you will have to return to the low-resolution display. You can do this by pressing the RUN/STOP and RESTORE keys simultaneously, resetting the Commodore to its state before the program was carried out.

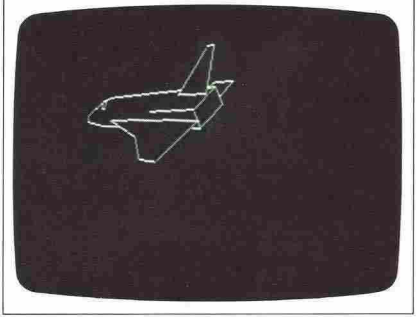
How to change the size of a display

You can alter the line machine program so that it produces the same display but at a different size. Try keying these lines into the program:

```
600  NX=NX/2 : NY=NY/2 : GT=ABS(NX-LX)
1090  IF PEN=0  THEN  LX=NX/2 : LY=NY/2
: GOTO 1030
```

This reduces the space shuttle display to half of its original size:

REDUCED LINE MACHINE DISPLAY



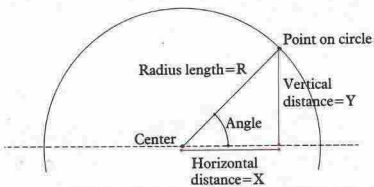
CURVES AND CIRCLES

Now that you know how to tackle graphics with straight lines, you can get your computer to draw some curves and circles. On the Commodore, there's no CIRCLE command to help you here, instead you need two new keywords, SIN and COS. Using these two commands you can produce some spectacular graphics with quite short programs.

How the Commodore draws a circle

If you sketch out part of a circle, you can relate each point on the circle to an angle at the circle's center.

CIRCLE CO-ORDINATES



You can write the distances X and Y in another way, as multiples of the angle and SIN or COS. Every angle has its own value of SIN and COS, and you can write the co-ordinates of any point on the circle like this:

R*COS(A),R*SIN(A)

Once you know this, you can start your Commodore on curves and circles. The next program produces a circle with a radius of 80 pixels. Remember to shift BASIC (if you haven't done so already) and LOAD the six high-resolution graphics subroutines before you RUN it:

CIRCLE PLOTTING PROGRAM

```
LIST

10 GOTO 1000
1000 COL=16 : POKE 53280,8 : GOSUB 100 :
 GOSUB 200
1010 FOR A=0 TO 2*π STEP π/120
1020 LX=80*COS(A)+160
1030 LY=80*SIN(A)+100
1040 GOSUB 400
1050 NEXT A
1060 GOTO 1060
READY.
```

Why computers don't work with degrees

In this first program, the angle has to vary from 0 to a full circle, which is 360 degrees. But as you will probably have noticed, there's no mention of 360 in the program – instead the loop runs from 0 to $2*\pi$, with an odd-looking STEP value of $\pi/120$. The reason for this is that the Commodore doesn't use degrees at all. Instead it measures angles in radians, just a different but more logical way of doing the same thing. A full-circle angle of 360 degrees is exactly equivalent to $2*\pi$ radians.

The symbol π (pronounced "pi") is a Greek letter that is produced on the Commodore by pressing SHIFT and the key next to RESTORE. It's an important mathematical constant, which has a value of 3.14159265... (you can see this by keying in PRINT π). This figure is the ratio of the length of a circle's circumference to its diameter. All you need to remember is that there are $2*\pi$ radians in a circle, so that $\pi/2$ radians are a quarter of a circle, $\pi/4$ an eighth and so on.

The all-purpose circle subroutine

You can now add another subroutine to the six that you already have so your Commodore can produce any circles you want. The subroutine is very similar to the previous program, except that instead of just plotting points around a circle, it draws lines between them to produce a complete outline.

This new subroutine begins at line 700. To use it, you need to give the computer three numbers. These are the values of the co-ordinates for the center of the circle, XC and YC, and the length, in pixels, of the circle's radius, RAD:

CIRCLE OUTLINE (SUBROUTINE 7)

```
LIST

700 A1=0 : A2=2*π
710 IF A1>=A2 THEN A2=A2+2*π : GOTO 710
720 DA=1+INT((A2-A1)/0.2)
730 A3=(A2-A1)/DA
740 LX=INT(RAD*COS(A1)+XC+0.5)
750 LY=INT(RAD*SIN(A1)+YC+0.5)
760 FOR A0=(A1+A3) TO A2 STEP A3
770 MX=INT(RAD*COS(A0)+XC+0.5)
780 MY=INT(RAD*SIN(A0)+YC+0.5)
790 GOSUB 600 : NEXT A0 : RETURN
READY.
```

When you have added this to the block of six subroutines that you already have, SAVE them together

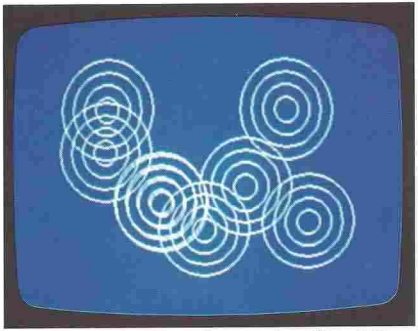
for use later. Your set of high-resolution graphics subroutines is now nearly complete.

Pattern design with the circle subroutine

If you select the co-ordinates of a circle's center at random, you can make the computer build up patterns:

CONCENTRIC CIRCLE PROGRAM

```
LIST

10 GOTO 1000
1000 COL=22 : POKE 53280,6 : GOSUB 100 :
 GOSUB 200
1010 FOR C=1 TO 8
1020 XC=50+INT(RND(0)*220)
1030 YC=50+INT(RND(0)*100)
1040 FOR RAD=10 TO 40 STEP 10
1050 GOSUB 700
1060 NEXT RAD : NEXT C
1070 GOTO 1070
READY.
```



Lines 1020 and 1030 produce a pair of X and Y co-ordinates at random so that neither is within 50 pixels of the screen edges. It is set like this so that the program can then draw circles up to a radius of 40. The loop at lines 1040 to 1060 repeatedly draws circles with the same center, but with gradually increasing radii. Line 1060 starts the whole process off again but with a new pair of random co-ordinates.

You can try altering the maximum radius of the circle and the STEP size between radii by changing the figures in line 1040 to:

```
1040 FOR RAD=10 TO 40 STEP 6
```

or even STEP 3, which draws smaller and tighter patterns.

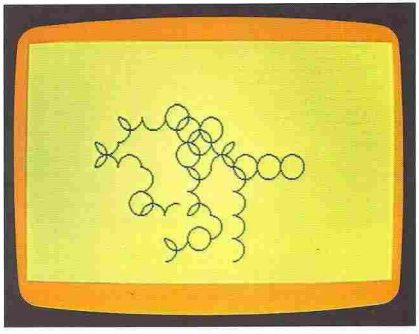
Programming wandering curves

When you keyed in the circle subroutine, you might have thought that it seemed more complicated than really necessary. You would have been quite right. In fact, not only is this subroutine capable of drawing circles, but it can also draw arcs, or parts of circles. To use this subroutine to draw an arc, you need to give the computer values for XC, YC and RAD just as for circles, but you also need to specify values for two extra variables, A1 and A2. These extra variables control the start and finish angles that the arc will be drawn between. These angles are measured in radians working clockwise from the positive X axis. When you use the subroutine to draw arcs you need to call it with GOSUB 710 instead of GOSUB 700.

Here is a program that uses GOSUB 710 to draw semicircles at random, each of which can go up, down, left or right across the screen. It does this by picking a random number from 1 to 4 inclusive, and then using this number to set the direction in which the semicircle will be drawn:

WANDERING CURVES PROGRAM

```
LIST

10 GOTO 1000
1000 COL=103 : RAD=10
1010 POKE 53280,8 : GOSUB 100 : GOSUB 20
0
1020 LX=160 : LY=100
1030 Q=INT(RND(0)*4)+1
1040 IF Q=1 THEN XC=LX : YC=LY-10 : A1=π
/2
1050 IF Q=2 THEN XC=LX+10 : YC=LY : A1=π

1060 IF Q=3 THEN XC=LX : YC=LY+10 : A1=3
*π/2
1070 IF Q=4 THEN XC=LX-10 : YC=LY : A1=0

1080 IF XC>300 OR XC<20 OR YC>180 OR YC<
20 THEN 1020
1090 A2=A1+π
1100 GOSUB 710
1110 GOTO 1030
READY.
```



FILLING SHAPES

So far you have built up quite a comprehensive graphics "toolkit" in the form of subroutines that will plot and draw a range of shapes. Now you can finish off the set by adding a last one which fills in the shapes you can draw with the others. The subroutine you'll use here can work with almost any closed shape to fill it with solid color. The word "closed" means that the shape must be bounded by a completely unbroken line of lit pixels – a dotted circle, for example, can't be filled in this way, but a drawn one can.

The shape-filling subroutine

To start you off, here is the subroutine which you should add to your stack of seven:

SHAPE FILLER (SUBROUTINE 8)

```
LIST -930

800 SP=0 : FU=0 : FD=0
810 FX=FX-1 : LX=FX : LY=FY : GOSUB 300
820 IF (PEEK(BYTE) AND MASK)=0 THEN 810
830 FX=FX+1 : LX=FX : GOSUB 400
840 LY=LY-1 : GOSUB 300
850 IF ((PEEK(BYTE) AND MASK)=0) AND (FU
=0) THEN FU=1 : GOSUB 940
860 IF ((PEEK(BYTE) AND MASK)<>0) AND (F
U=1) THEN FU=0
870 LY=LY+2 : GOSUB 300
880 IF ((PEEK(BYTE) AND MASK)=0) AND (FD
=0) THEN FD=1 : GOSUB 940
890 IF ((PEEK(BYTE) AND MASK)<>0) AND (F
D=1) THEN FD=0
900 LY=LY-1 : LX=LX+1 : GOSUB 300
910 IF (PEEK(BYTE) AND MASK)=0 THEN 830
920 IF SP=0 THEN RETURN
930 GOSUB 970 : GOTO 810

READY.
```

```
LIST 940-

940 IF SP>9 THEN PRINT "SHAPE TOO COMPLE
X TO FILL" : END
950 ST(SP,0)=LX : ST(SP,1)=LY : SP=SP+1
960 RETURN
970 FU=0 : FD=0 : SP=SP-1
980 FX=ST(SP,0) : FY=ST(SP,1)
990 RETURN
READY.
```

To use this routine, you need to set two variables, FX and FY. These are any co-ordinates within the area that you want to fill – it doesn't matter where in the area they are. You can then call the subroutine with GOSUB 800 and it will fill in the area you have selected.

Now for a word of warning. By this stage, you will have seen that the Commodore plots and draws quite slowly with BASIC. The filling routine is no faster than the drawing routine, and because there are so many more pixels to deal with, shape filling takes a long time. However, as you will see, the results make the waiting worthwhile.

How the shape filler works

Although the shape filler looks complicated, what it actually does is quite simple. Lines 810 and 820 move the position of the variable FX to the left from its starting point until the computer finds a boundary. Lines 830 to 890 then move back from left to right, plotting pixels as they go to fill in a horizontal line. While pixels are being plotted from left to right, lines 850 to 890 are also checking the pixels above and below to see if they will have to be plotted later.

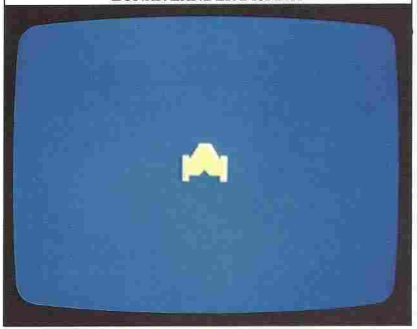
If pixels need to be plotted above and below the current line, then the co-ordinates of the left-hand ends of these lines of pixels will be stored by program lines 940 to 960. The co-ordinates are put into an "array", a method for storing information so that each item can be retrieved separately (arrays are explained on pages 52–53). When the end of the current line has been reached (that is, when the routine has encountered a boundary line on the right), a pair of X,Y co-ordinates is recovered from the array and the process begins again, starting from these new co-ordinates.

To see the shape filler in action, key the following program onto the end of the completed set of eight graphics subroutines, and then RUN it. Remember as usual that you will need to shift BASIC if you haven't done so already:

FILLING A LUNAR LANDER

```
LIST
10 GOTO 1000
1000 COL=118 : GOSUB 100 : GOSUB 200 : P
OKE 53280,6
1010 LX=155 : LY=80
1020 FOR C=1 TO 17
1030 READ MX,MY : GOSUB 600
1040 NEXT C
1050 FX=160 : FY=90 : GOSUB 800
1060 GOTO 1060
1070 DATA 165,80,175,95,175,90
1080 DATA 180,90,180,110,175,110
1090 DATA 175,105,165,105,160,100
1100 DATA 155,105,145,105,145,110
1110 DATA 140,110,140,90,145,90
1120 DATA 145,95,155,80
READY.
```

LUNAR LANDER DISPLAY



In this program, line 1000 sets the foreground color to yellow and the background to blue, switches the machine to high resolution and clears the screen. Lines 1010 to 1040 use the six lines of DATA at the end of the program as co-ordinate pairs and draw the outline of the lunar lander. Line 1050 then sets FX and FY to a point within the outline. Line 1060 is only included to prevent text messages spoiling the display at the end.

Filling in the space shuttle display

The next program takes the space shuttle from page 19, fills it in and then adds a moon to it (this is programmed by lines 1120 to 1150). The co-ordinates between lines 1110 and 1116 determine which parts of the shuttle are to be filled in. In this program, there are four co-ordinate pairs specified – one for the crew's observation window, two for the wings and one for the tail. The moon filling is specified by line 1160. It's drawn in a different ink color, set in line 1120. Again, the program finishes with an endless loop:

SPACE SHUTTLE FILLER

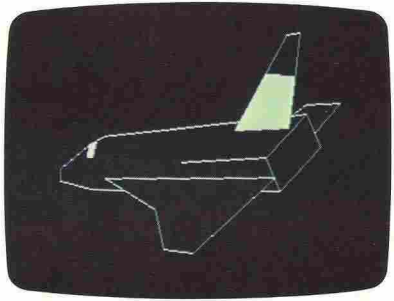
```
LIST -1140

10 GOTO 1000
1000 GOSUB 100
1010 COL=208 : GOSUB 200
1020 PEN=0 : LX=0 : LY=0
1030 READ NX,NY
1040 IF NX>=0 THEN 1090
1050 IF NY=1 THEN PEN=1 : GOTO 1030
1060 IF NY=0 THEN PEN=0 : GOTO 1030
1070 IF NY=2 THEN 1110
1080 COL=-NY : GOTO 1030
1090 IF PEN=0 THEN LX=NX : LY=NY : GOTO
1030
1100 GOSUB 600 : GOTO 1030
1110 FX=73 : FY=110 : GOSUB 800
1112 FX=220 : FY=80 : GOSUB 800
1114 FX=173 : FY=194 : GOSUB 800
1116 FX=200 : FY=70 : GOSUB 800
1120 COL=112 : XC=60 : YC=40 : RAD=20
1130 A1=4.9 : A2=3 : GOSUB 710
1140 XC=40 : YC=20 : RAD=20

READY.
```

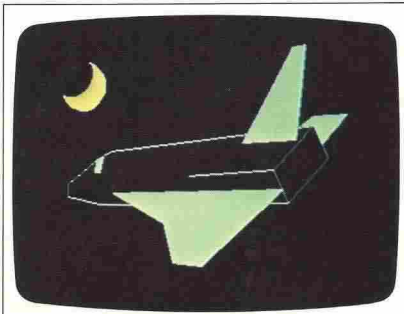
SPACE SHUTTLE FILLER

```
LIST 1150-

1150 A1=5.8 : A2=2 : GOSUB 710
1160 FX=70 : FY=50 : GOSUB 800
1170 GOTO 1170
1200 DATA -1,0,49,123,-1,1,74,102,73,112
,78,112,81,98,74,102,88,94,212,78
1210 DATA 207,85,254,4,266,2,256,82,207,
85,-1,0,258,72,-1,1
1220 DATA 272,70,286,98,250,137,236,108,
272,70
1230 DATA 258,72,272,62,304,60,280,86,-1
,0,245,125,-1,1
1240 DATA 90,130,136,156,145,192,173,195
,245,125
1250 DATA -1,0,236,108,-1,1,158,114,-1,0
,106,140,-1,1,78,140,49,134,49,123
1260 DATA -1,0,250,137,-1,1,233,137
1360 DATA -1,2
READY.
```

Here are two displays of the shuttle filler in action. In the first one, the fill is still under way, in the second, it's complete:

FILLING THE SPACE SHUTTLE





NATURAL GRAPHICS

On pages 20-21, you saw how the Commodore's BASIC commands SIN and COS can produce circles and arcs. These are not the only uses to which you can put these two functions. If you take a look back at the first program on page 20, the one that plotted a circle with dots, you can make just a few small changes to produce quite different results. To do this you will need all the eight high-resolution subroutines again, so if they are not already in memory, LOAD them into your computer before you start.

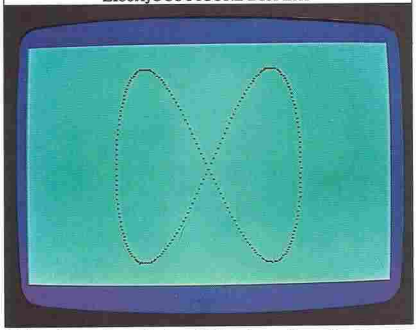
How to throw a circle out of step

The circle program on page 20 produces a circle because the X and Y co-ordinates vary in exactly opposite ways. When X is zero, Y is at its maximum value and vice versa. What happens if you deliberately make them vary at different rates? Make just one change to the program – alter the angle after the SIN command by changing line 1030 to:

```
1030 LY=80*SIN(2*A)+100
```

Here's the display it produces (you can set the colors to any combination you like):

LISSAJOUS FIGURE DISPLAY



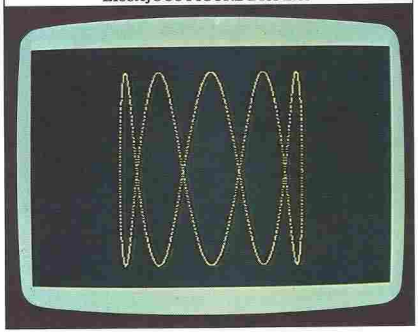
This shape is called a "Lissajous figure" after the French physicist who first investigated them. The number of loops in the display depends on how many times the angle after SIN has been multiplied.

You can adapt the circle program to produce an infinite variety of results like this. Here's another way of changing it:

```
1010 FOR A=0 TO 2*π/720
1030 LY=80*SIN(5*A)+100
```

This time the angle has been multiplied by 5, as you can tell by looking at the display:

LISSAJOUS FIGURE DISPLAY



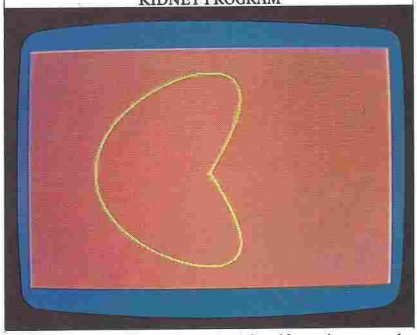
Programming some complex curves

Now you can make a different sort of change to the program. Try keying in these lines:

```
1010 R=0: FOR A=0 to 2* π STEP π/480
1020 LX=(R*COS(A)*R*SIN(0.5*A))+160
1030 LY=8*R*SIN(A)+100
```

Here's what it looks like if you RUN it after changing colors as well:

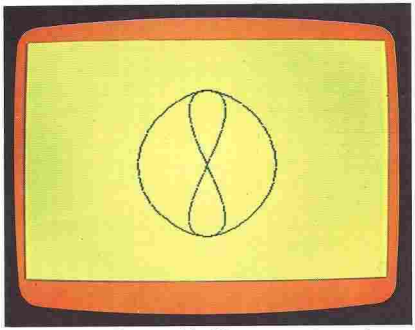
KIDNEY PROGRAM



The program will continue drawing if you increase the range of angle values. The next program roughly doubles the range. It also increases the RUNning speed by drawing short lines instead of plotting individual points. Each new position calculated is the starting point for the line to be drawn on the next pass around the FOR ... NEXT loop:

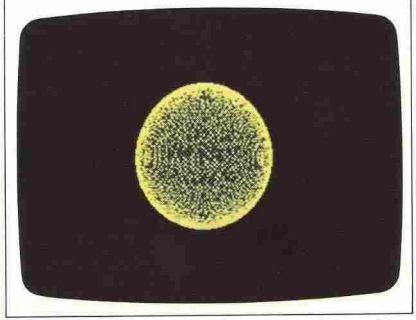
HOURGLASS PROGRAM

```
LIST

10 GOTO 1000
1000 COL=7 : POKE 53280,2 : GOSUB 100 :
GOSUB 200
1010 R=60
1020 X=0 : Y=0 : LX=160 : LY=100
1030 FOR A=0.2 TO 12.6 STEP 0.2
1040 I=R*COS(A)*SIN(0.5*A)
1050 J=R*SIN(A)
1060 NX=LX+I-X : NY=LY+J-Y
1070 GOSUB 600
1080 X=I : Y=J : NEXT A
1090 GOTO 1090
READY.
```



You can experiment with this program to produce a whole variety of more complicated shapes. Here's one which has a long RUNning time:

BALL DISPLAY



This design is produced by changing colors again, and by altering lines 1030 and 1040 so that they now read:

```
1030 FOR A=0.001 TO 1000 STEP 0.1
1040 I=R*COS(A)*SIN(0.98*A)
```

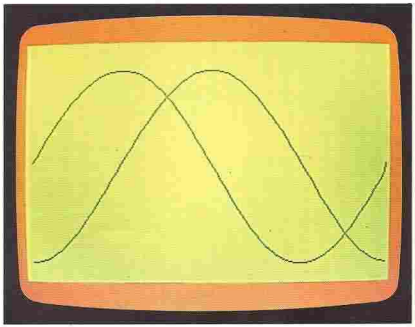
Here SIN and COS are working on very slightly different angles. The shape starts off by being quite open, but very gradually the computer fills it in until it becomes the filled ball on the previous screen. The patterns it produces as it develops are just as interesting as the final design itself. Again, you can try changing these two new lines to produce different shapes.

How to draw a graph of SIN and COS

As a final example of natural graphics, here is a program which actually shows you how the values of SIN and COS vary against each other for angles between 0 and 2* π radians – 0 on the left-hand side and 2*π on the right. The values of SIN and COS both vary between −1 and +1; in this program they are exaggerated so that you can see the variation more easily:

SIN AND COS WAVE PROGRAM

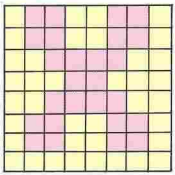
```
LIST

10 GOTO 1000
1000 COL=151 : POKE 53280,8 : GOSUB 100
: GOSUB 200
1010 LX=3 : LY=95
1020 FOR A=π/36 TO 2*π STEP π/36
1030 NX=INT(A*50)
1040 NY=INT(95-80*SIN(A))
1050 GOSUB 600
1060 NEXT A
1070 LX=3 : LY=15
1080 FOR A=π/36 TO 2*π STEP π/36
1090 NX=INT(A*50)
1100 NY=INT(35-80*COS(A))
1110 GOSUB 600
1120 NEXT A
1130 GOTO 1130
READY.
```



DESIGNING CHARACTERS

If you look closely at the screen, you can see the pixel pattern that makes up each symbol or character. Each of the characters is made by displaying a different arrangement of pixels within an 8×8 grid. The letter X, for example, looks like this:

SINGLE CHARACTER



These pixel patterns are stored in an area of ROM so that the character pattern is ready for use as soon as the Commodore is switched on. Inside this "character generator" ROM, each character pixel is represented by a single bit of memory. An 8×8 grid therefore needs 8 bytes of memory, so that an entire character set of 256 items takes up 2K (2048 bytes). With the Commodore, you can design your own characters, replacing the whole set if you want to.

How to change the character patterns

Given that all the pixel patterns are stored in ROM, you may be wondering how you can change them at all. What you have to do is point the Commodore to a different location when it is looking for the pixel pattern information. If you tell the Commodore to get the patterns from RAM instead of ROM, you can then POKE in your own characters. This is fairly easy to do. However, if you do this, your Commodore will "forget" all its built-in characters. So if you want it to remember some of them, you have to copy the characters you want to keep, such as the numbers and letters, from ROM into RAM before you switch over.

The next program takes copies of 64 characters – the numbers and letters – from ROM into RAM, and then switches the machine to using the RAM characters. You'll notice that the RAM characters are stored from location 2048 upward. This number should be familiar to you by now – it's the location that BASIC normally uses for its programs, so that you will have to move the bottom of BASIC as described on page 14 before you RUN the program.

Because the computer has to transfer a total of 512 bytes to new memory locations, it takes a few seconds before the process is complete and the selected ROM characters are stored in the specified area of RAM:

SWITCHING TO RAM CHARACTERS

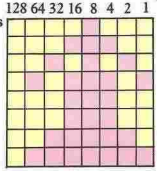
```
LIST

10 POKE 56334,PEEK(56334) AND 254
20 POKE 1,PEEK(1) AND 251
30 FOR A=0 TO 511
40 POKE 2048+A,PEEK(53248+A) : NEXT A
50 POKE 1,PEEK(1) OR 4
60 POKE 56334,PEEK(56334) OR 1
70 POKE 53272,(PEEK(53272) AND 240)+2
READY.
```

When you RUN this, you should soon be able to tell if it is working because the normal block cursor, which is not copied from ROM to RAM, will be replaced by a flashing dot pattern.

Making and storing your own symbols

Say you want to put a small symbol on the screen – a rocket for a space game. The first thing you need to do is draw the character. It's the same technique as designing a sprite, but much simpler.

SINGLE CHARACTER DESIGN



As usual, each pixel is either "on" or "off", and this is shown by filling in the squares.

To convert this to a sequence of bytes for storage, you need to take each row of 8 pixels in turn, and add their values to form a single byte. Following this procedure, the single character is converted into 8 bytes.

These now need to be POKEd into memory onto the end of the existing character data. The best characters to redefine are the shifted characters, SHIFT A to SHIFT Z. To add the rocket design to the character set, type the following lines onto the end of the previous program:

ROCKET CHARACTER PROGRAM

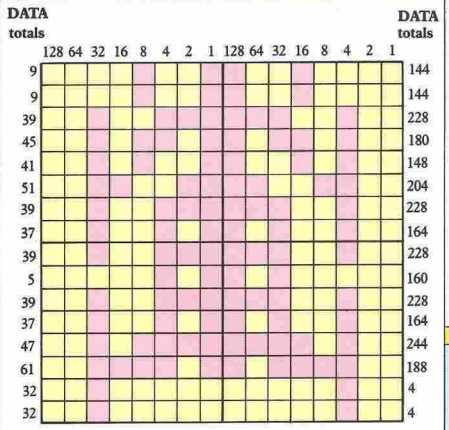
```
LIST

80 PRINT CHR$(147)
90 PRINT "   ";CHR$(97)
100 FOR T=1 TO 1000 : NEXT T
110 FOR C=0 TO 7
120 READ BYTE
130 POKE C+2568,BYTE
140 NEXT C
150 DATA 8,28,42,93,28,28,62,127
READY.
```

When you RUN this program you will see the current CHR$(97) character appear on the screen, and after a short time it will change to the one that you have just defined, the rocket.

Adding your own characters together

The first thing you will notice is that the rocket is extremely small. This is because it only occupies the same amount of space as a single letter on the screen. But although user-defined characters are based on an 8×8 grid, there is no reason why your designs should not cover more than one grid, so you can make them any size or shape you want. Here's a more complex design:

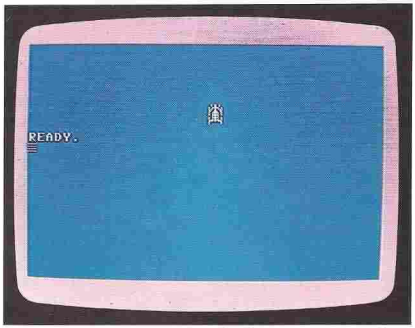
MULTI-CHARACTER DESIGN



In the next program, these four characters are stored as CHR$(97) to CHR$(100). Here's the complete program and the display it produces:

FOUR-CHARACTER PROGRAM

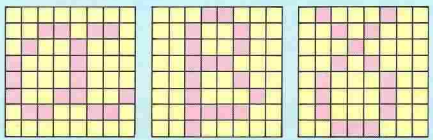
```
LIST

200 PRINT CHR$(147) : POKE 53280,4 : POK
E 53281,6
210 FOR C=0 TO 31
220 READ BYTE : POKE C+2568,BYTE
230 NEXT C
240 X=20 : Y=5
250 POKE 214,Y : PRINT : POKE 211,X
260 PRINT CHR$(97);CHR$(98)
270 POKE 214,Y+1 : PRINT : POKE 211,X
280 PRINT CHR$(99);CHR$(100)
300 DATA 4,4,19,22,20,25,19,18,200
310 DATA 200,242,218,202,230,242,210
320 DATA 19,2,19,18,23,30,16,16
330 DATA 242,208,242,210,250,222,2,2
READY.
```



This program assumes that the first program to place the ROM characters into RAM has already been RUN. It is also written so that the rocket can be positioned anywhere on the screen by changing the values of X and Y on line 240. These values control the position used to display CHR$(97), the top left part of the rocket.

Because there is room to define a complete set of 256 characters if you want to, you can draw up more than one completely new alphabet and still have enough memory locations left to keep the Commodore's pre-programmed letter and number symbols.

USER-DEFINED LETTERS



ADVANCED SPRITEMAKING

A standard sprite is 24×21 pixels, but can be expanded by a factor of 2 in both horizontal and vertical directions. This means that a fully expanded sprite occupies 48×42 pixels, a fourfold increase in area. These variations in size do not mean that you have to put extra DATA into the program – locations V+23 and V+29 do all the work for you.

V+29 controls expansion horizontally, and V+23 controls expansion vertically. The bits in both these bytes each control a separate sprite, so that to expand sprite 0, you would need to set bit 0 in either or both locations to 1. For sprite 1, you would need to set bit 1, and so on. So for sprite 0, expansion in both directions would be activated by:

```
POKE V+23,1:POKE V+29,1
```

To expand sprite 1, you would need to set bit 1. This bit has a numeric value of 2, so to expand sprite 1 in both directions you would need to POKE a value of 2 into both locations. To expand a combination of sprites, you simply add all the POKE values together. POKE V+23,3 would expand both sprites 0 and 1 vertically.

When you use either of these expansion locations, all that changes is the size that a pixel is plotted. This means that when you enlarge a sprite, you don't get any increase in resolution. The sprite will be larger but coarser. The next program allows you to compare all the possible expansions. It stores sprites in the area normally used by BASIC, so before you key it in, you will need to move the BASIC storage area by the method shown on page 14.

The program makes sprites 0, 1, 2 and 3 all from the same DATA. Next the program gives each sprite one of the four different sizes available. The first screen is the sprite control section:

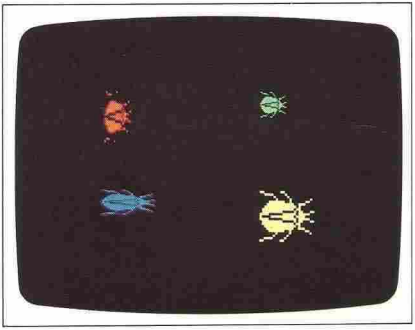
SPRITE EXPANSION PROGRAM

```
LIST -180

10 PRINT CHR$(147)
20 V=53248
30 POKE 53280,0 : POKE 53281,0
40 FOR C=0 TO 62
50 READ BYTE : POKE 2048+C,BYTE
60 NEXT C
70 POKE 2040,32 : POKE 2041,32
80 POKE 2042,32 : POKE 2043,32
90 POKE V+39,2
100 POKE V+40,5
110 POKE V+41,6
120 POKE V+42,7
130 POKE V,60 : POKE V+1,60
140 POKE V+2,200 : POKE V+3,60
150 POKE V+4,60 : POKE V+5,140
160 POKE V+6,200 : POKE V+7,140
170 POKE V+23,9
180 POKE V+29,12

READY.
```

The second screen shows the sprite DATA which is used four times to make up the display:

SPRITE EXPANSION PROGRAM

```
LIST 190-

190 POKE V+21,15
200 GOTO 200
400 DATA 192,96,0,60,16,0,2,8,0
410 DATA 1,4,254,15,228,16,31,254,112
420 DATA 63,255,64,127,254,132,127,193,9
9
430 DATA 240,63,124,135,254,240,240,63,1
24
440 DATA 127,193,99,127,254,192,63,255,6
4
450 DATA 31,254,112,15,228,12,1,4,2
460 DATA 2,8,0,60,16,0,192,96,0,0
READY.
```



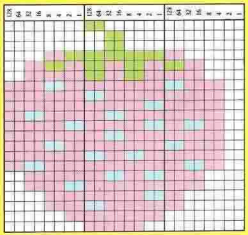
SPRITE MULTI-COLOR BIT VALUES

Each pair of pixels in a sprite row is treated as one unit. By using the bit values in this table for each pair, you can set the pair to any of four colors, each controlled by a different register.

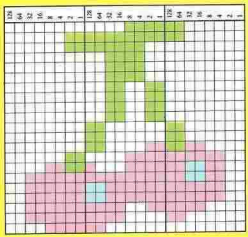
Bit pair value	Register controlling pixel pair color
00	Screen color register (53281)
01	Sprite multi-color register 1 (V+37)
10	Normal sprite color registers (V+39–V+46)
11	Sprite multi-color register 2 (V+38)

MULTI-COLOR SPRITE DESIGNS

Here two multi-color sprites have been drawn up and their DATA values calculated. The bit value for each pair of pixels is set by using the system shown in the table on the previous page.



0	64	0
0	16	0
0	16	0
9	85	128
26	102	96
42	102	160
174	170	232
170	234	168
170	171	168
186	186	168
171	171	184
170	186	168
174	170	232
170	234	168
186	170	184
42	187	160
43	170	160
10	171	128
10	234	128
2	170	0
0	168	0



0	85	64
1	85	64
1	84	0
0	20	0
0	20	0
0	20	0
0	17	0
0	17	0
0	17	0
0	17	0
0	64	64
0	64	64
0	66	96
1	10	168
9	138	184
42	170	184
42	234	168
42	234	168
42	162	160
42	160	128
10	128	0

Programming multi-color sprites

In addition to the enlarging that is available with sprites, you can also make up a sprite with a mixture of colors. Here's a listing which produces the sprites shown above:

MULTI-COLOR SPRITE PROGRAM

```
LIST -180

10 PRINT CHR$(147)
20 POKE 53280,0 : POKE 53281,0
30 V=53248
40 FOR C=0 TO 127
50 READ BYTE : POKE 2048+C,BYTE
60 NEXT C
70 Y1=0 : Y2=0
80 D1=1 : D2=1.1
90 POKE V+23,3 : POKE V+29,3 : POKE 2040
,32
100 POKE 2041,33
110 POKE V+39,2
120 POKE V+40,5
130 POKE V+37,5
140 POKE V+38,6
150 POKE V+28,3
160 POKE V+21,3
170 POKE V,100 : POKE V+1,Y1
180 POKE V+2,150 : POKE V+3,Y2

READY.
```

MULTI-COLOR SPRITE PROGRAM

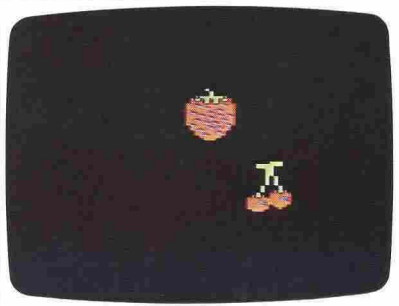
```
190 IF Y1>250 THEN Y1=0
200 IF Y2>250 THEN Y2=0
210 Y1=Y1+D1 : Y2=Y2+D2
220 GOTO 170
300 DATA 0,64,0,0,16,0
310 DATA 0,16,0,9,85,128
320 DATA 38,102,96,42,102,160
330 DATA 174,170,232,170,234,168
340 DATA 170,171,168,186,186,168
350 DATA 171,171,184,170,186,168
360 DATA 174,170,232,170,234,168
370 DATA 186,170,184,42,187,160
380 DATA 43,170,160,10,171,128
390 DATA 10,234,128,2,170,0
400 DATA 0,168,0,0
410 DATA 0,5,64,1,85,64,1,84,0,0,20,0
420 DATA 0,20,0,0,20,0,0,17,0,0,17,0
430 DATA 0,17,0,0,17,0,0,64,64,0,64,64
440 DATA 0,66,96,1,10,168,9,138,184
450 DATA 42,170,184,42,234,168
460 DATA 42,234,168,42,162,160
470 DATA 42,160,128,10,128,0,0
READY.
```

As you have already seen, a single color sprite has a horizontal resolution of 24 pixels. If you use the multi-color facility, the horizontal resolution drops to 12. This does not mean that the sprites drop to half width, because each of the pixels in a multi-color sprite is displayed twice as wide on the screen.

In a normal sprite each bit controls one pixel on the screen. In a multi-color sprite, each pair of bits in the bit pattern controls the color of one pair of pixels on the screen. If a pair of bits has the binary value 00, then the pixels on the screen will be transparent, allowing the background to show through. If the pair has the value 10, then the normal sprite color register is used to determine the pixels' color. The difference comes when a pair of bits has the value 01 or 11, because then the color of the pixels is taken from one of two special multi-color registers at VIC locations V+37 and V+38. This means that each sprite can contain up to four colors.

Here is the display that the multi-color sprite program produces:

MULTI-COLOR SPRITE DISPLAY



SPRITE ANIMATION

When you are programming sprites, you need to decide where to put the sprite DATA in the Commodore's memory. As you may have seen in Book 1, there is enough space for three sprites in the Commodore's cassette input/output buffer starting at locations 832, 896 and 960. This area can only be used if you don't want to use the cassette at the same time that the sprite program is RUNning, and if you don't want to use more than three sprites at the same time. However, there are places in the Commodore's memory where you can store much more sprite DATA than this. You used one of these areas on the previous two pages by moving the BASIC storage area from its normal position.

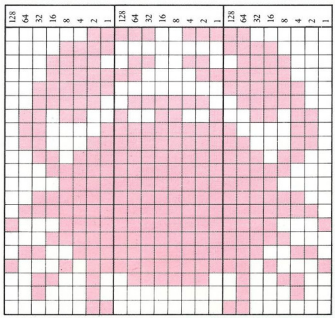
Where to store sprites

By making a large section of memory available for holding your sprites, you can store enough information to program 32 of them from locations 2048 upward – four times as many sprites as you can have on the screen at once. This means that you can have a large store of sprite shapes that can be called on very quickly with interesting results.

To see what you can do with this, here is a single sprite design and a program which puts the DATA into the memory area in which BASIC is normally held. Once you understand how this sort of program works, you can try one which modifies the sprite as it moves:

CRAB SPRITE DESIGN

Bit values



DATA values

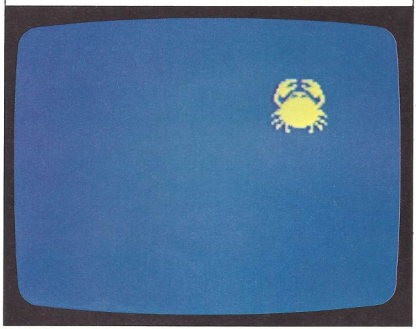
3	231	192
15	0	240
30	102	120
31	195	248
63	0	252
62	126	124
108	153	54
97	255	134
99	255	198
55	255	236
31	255	248
15	255	240
63	255	252
127	255	254
143	255	241
31	255	248
103	255	230
139	255	209
50	126	76
34	0	68
3	0	192

Because the program uses the space normally occupied by BASIC, remember to shift the BASIC storage area before keying it in (if you've forgotten how to do this, look back at page 14). When you RUN the program you'll see that the single sprite shape moves around the screen. It's controlled by changing the X and Y co-ordinates by 2 on each cycle through a loop:

CRAB SPRITE PROGRAM

```
LIST -180

10 PRINT CHR$(147) : V=53248
20 FOR C=12 TO 62 : READ BYTE
30 POKE 2048+C,BYTE : POKE 2112+C,BYTE
40 POKE 2176+C,BYTE : NEXT C
50 FOR C=0 TO 2 : FOR K=0 TO 11
60 READ BYTE : POKE 2048+64*C+K,BYTE
70 NEXT K : NEXT C : POKE 53280,0 : POKE
 53281,0
80 POKE V+39,7 : POKE V+21,1
90 X=100 : DX=2 : S=32 : DS=-0.5
100 T=INT(X/256) : POKE 2040,S : POKE V+
23,3 : POKE V+29,3
110 POKE V+16,T
120 POKE V,X-T*256
130 POKE V+1,80
140 POKE 2040,INT(S)
150 IF S=32 OR S=34.5 THEN DS=-DS
160 IF X<50 OR X>280 THEN DX=-DX
170 X=X+DX : S=S+DS
180 GOTO 100

READY.
```

```
LIST 190-

200 DATA 3,231,192,15,0,240
210 DATA 30,102,120,31,195,248
220 DATA 63,0,252,62,126,124
230 DATA 108,153,54,97,255,134
240 DATA 99,255,198,55,255,236
250 DATA 31,255,248,15,255,240
260 DATA 63,255,252,127,255,254
270 DATA 143,255,241,31,255,248
280 DATA 103,255,230,139,255,209
290 DATA 50,126,76,34,0,68
300 DATA 3,0,192
READY.
```



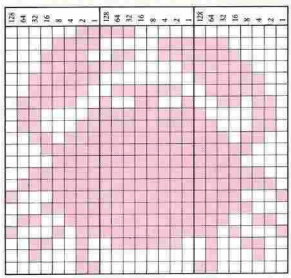
Ways to change sprite shapes

Now it is possible to get some action into the sprite itself as well as moving it around. You can do this by setting up two or more sprite areas in memory so that they contain images of the same sprite in different positions. Locations 2040 to 2047 tell the Commodore's VIC chip where in memory it can find the sprite DATA for sprites 0 to 7. All you need to do to make a sprite change shape is to switch the VIC chip quickly around a sequence of images. It's like making a cartoon with numbers instead of with pencil and paper. It is done by POKEing the memory location controlling where the VIC chip gets a sprite's DATA from with a sequence of numbers that point to different sets of sprite DATA.

Here is an example which shows you how this is done. Staying with the crab sprite, you can define some more versions of it. The designs at the bottom of this page show the crab with its claws in different positions. Having looked at these designs, you can then key in the program which uses this DATA to animate the crab.

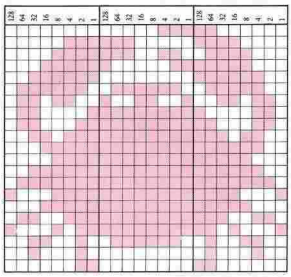
One thing you should notice here is that the program does not contain the complete set of DATA for the three different versions of the crab. This is because most of the DATA for each design is the same, so to store all the DATA would waste a lot of memory. Instead, only the top four lines of each crab are stored separately.

ANIMATED SPRITE DESIGNS



DATA values

3	224	0
15	7	224
30	103	240
31	193	248
63	0	252
62	126	124
108	153	54
97	255	134
99	255	198
55	255	236
31	255	248
15	255	240
63	255	252
127	255	254
143	255	241
31	255	248
103	255	230
199	255	209
50	126	76
34	0	68
3	0	192



DATA values

0	7	192
7	224	240
15	230	120
31	131	248
63	0	252
62	126	124
108	153	54
97	255	134
99	255	198
55	255	236
31	255	248
15	255	240
63	255	252
127	255	254
143	255	241
31	255	248
103	255	230
199	255	209
50	126	76
34	0	68
3	0	192

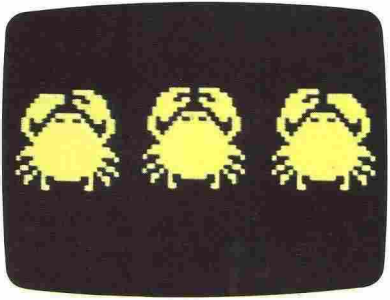
ANIMATED CRAB SPRITE PROGRAM

```
LIST -180

10 PRINT CHR$(147)
20 V=53248
30 POKE 53280,6 : POKE 53281,6
40 FOR C=0 TO 62
50 READ BYTE : POKE 2048+C,BYTE
60 NEXT C
70 X=100 : Y=100
80 DX=1 : DY=2
90 POKE 2040,32 : POKE V+23,1 : POKE V+2
9,1
100 POKE V+39,7 : POKE V+21,1
110 T=INT(X/256)
120 POKE V+16,T
130 POKE V,X-T*256
140 POKE V+1,Y
150 IF X<60 OR X>280 THEN DX=-DX
160 IF Y<60 OR Y>180 THEN DY=-DY
170 X=X+DX : Y=Y+DY
180 GOTO 110

READY.
```

```
LIST 190-

200 DATA 63,0,252,62,126,124
210 DATA 108,153,54,97,255,134
220 DATA 99,255,198,55,255,236
230 DATA 31,255,248,15,255,240
240 DATA 63,255,252,127,255,254
250 DATA 143,255,241,31,255,248
260 DATA 103,255,230,135,255,209
270 DATA 50,126,76,34,0,68
280 DATA 3,0,192
290 DATA 3,224,0,15,7,224
300 DATA 30,103,240,31,193,248
310 DATA 3,231,192,15,0,240
320 DATA 30,102,120,31,195,248
330 DATA 0,7,192,7,224,240
340 DATA 15,230,120,31,131,248
READY.
```

Now as the crab moves along, its claws open and shut like those of a cartoon figure, as you can see in these close-ups of the program RUNning:

ANIMATED CRAB SPRITE DISPLAYS



OVERLAPS AND COLLISIONS

What happens when a sprite moves into a position on the screen already occupied by something else? If you have RUN a sprite program while leaving text on the screen, you will probably have found that the sprite seems to "float" over the text, covering it up but not erasing it. On the other hand when two sprites meet, the results can be much more varied.

Setting overlaps with sprite priorities

On the Commodore you can tell the computer how to overlap sprites and background, and you can also make it take specified action if it detects a collision.

When two sprites meet, there is a simple rule which controls which is "in front" and which is "behind". The lower-numbered sprite will always appear to pass in front of the higher-numbered one. This gives sprite 0 the highest priority while sprite 7 has the lowest. This means that if, for example, sprite 2 meets sprite 4, sprite 2 will appear to pass in front of sprite 4. So sprite/sprite priorities are just a matter of careful choice of sprite numbers.

When a sprite meets a background object, the situation is a little different. The memory location 53275 (or V+27 in the VIC chip) controls the sprite-background display priorities. Each bit controls the priority for one sprite – bit 0 controlling sprite 0, bit 1 sprite 1 and so on. If you set a bit to 1 in this byte, then in a collision the corresponding sprite will pass behind any background it encounters. Conversely, if a bit is reset to 0, then its sprite will pass in front of any background object.

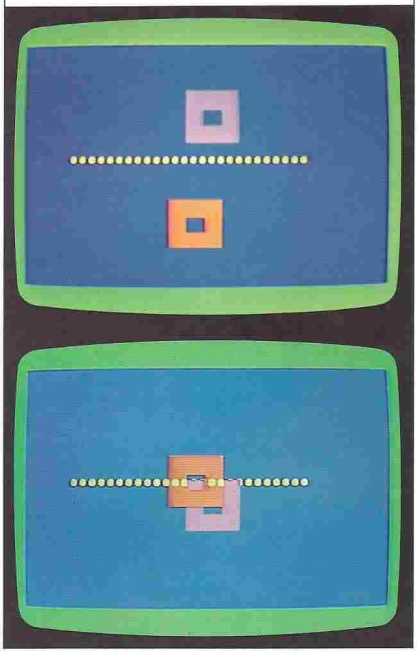
You can use these techniques to set up some interesting effects. The following program shows sprite priorities at work. It creates a situation which is impossible in reality but easy with sprites:

SPRITE PRIORITIES PROGRAM

```
LIST -200

10 PRINT CHR$(147);CHR$(158)
20 POKE 53280,5 : POKE 53281,6
30 V=53248
40 FOR C=0 TO 62
50 READ BYTE : POKE 2048+C,BYTE
60 NEXT C
70 POKE 2040,32 : POKE 2041,32
80 POKE V+27,1
90 POKE V+39,8 : POKE V+40,4
100 POKE 214,10 : PRINT
110 FOR X=5 TO 30
120 POKE 211,X : PRINT "@";
130 NEXT X
140 POKE V+21,3:POKE V+23,3:POKE V+29,3
150 Y1=60 : Y2=200
160 D1=6 : D2=-6
170 POKE V,150 : POKE V+2,165
180 POKE V+1,Y1 : POKE V+3,Y2
190 IF Y1<60 OR Y1>200 THEN D1=-D1
200 IF Y2<60 OR Y2>200 THEN D2=-D2

READY.
```

SPRITE PRIORITIES PROGRAM AND DISPLAYS

```
LIST 210-

210 Y1=Y1+D1 : Y2=Y2+D2
220 FOR T=1 TO 200 : NEXT T : GOTO 180
230 DATA 255,255,255,255,255,255
240 DATA 255,255,255,255,255,255
250 DATA 255,255,255,255,255,255
260 DATA 255,255,255,255,255,255
270 DATA 255,0,255,255,0,255
280 DATA 255,0,255,255,0,255
290 DATA 255,0,255,255,0,255
300 DATA 255,255,255,255,255,255
310 DATA 255,255,255,255,255,255
320 DATA 255,255,255,255,255,255
330 DATA 255,255,255
READY.
```



In this program there are two sprites, 0 and 1. Sprite 0 has priority. What makes the program interesting is the

way that the priorities are set up between the sprites and the background. Sprite 1 has priority over the background, while sprite 0 is made to move behind it, creating a strange effect on the screen.

Detecting collisions on the screen

The other important facility provided by the Commodore with moving sprites is collision detection. The computer considers a collision to have occurred if any lit pixel in a moving sprite is drawn at the same screen position as a lit pixel from any other object. This means that for programs with missiles and targets for example, it doesn't matter how complex the missile or target shapes are – the programming does not get any more involved.

There are two types of collision possible on the screen, one where a sprite hits another sprite and one where a sprite hits a background object. Both these situations are dealt with in much the same way, with individual bits being set within the VIC chip registers to "flag" the fact that a collision has taken place.

Two VIC registers are involved in this, V+30 and V+31. The first detects sprite/sprite collisions, and the second detects sprite/background collisions. Both of these two registers are split into individual bits, each controlling one sprite in the usual way. To start off with, the VIC chip resets these registers to zero, and then when any collisions take place, the appropriate bits are set to 1, and the byte is held like that until the VIC register is read with PEEK, which resets it back to zero.

This means that your software has a chance to detect any collision, no matter how momentary it may be. However, you should remember that reading the registers always resets them to zero, so the best thing to do is to read the values into a variable so that you don't lose a value before you have finished with it. The next program is similar to the last one except that this time it detects and signals collisions:

SPRITE COLLISION PROGRAM

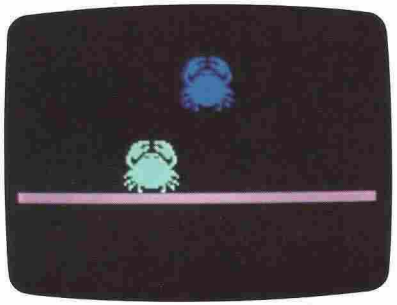
```
10 S=54272 : POKE S+24,15
20 POKE S+1,115 : POKE S,88
30 POKE S+5,0 : POKE S+6,240
40 PRINT CHR$(147);CHR$(156)
50 POKE 53280,0 : POKE 53281,0
60 V=53248
70 FOR C=0 TO 62
80 READ BYTE : POKE 2048+C,BYTE
90 NEXT C
100 POKE 2040,32 : POKE 2041,32
110 POKE V+27,2 : POKE V+23,3 : POKE V+2
9,3
120 POKE V+39,3: POKE V+40,6
130 POKE 214,12 : PRINT
140 PRINT CHR$(18);" "

150 POKE V+21,3
160 X=120 : Y=130
170 D1=3 : D2=-1.5
180 POKE V,X+D1 : POKE V+2,170
190 POKE V+1,135 : POKE V+3,Y
200 POKE S+4,16 : POKE V+27,(D1+3)/2

READY.
```

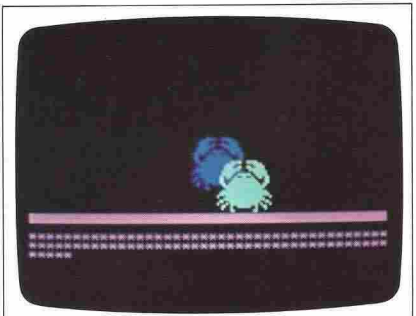
SPRITE COLLISION PROGRAM

```
LIST 210-

210 IF X<80 OR X>200 THEN D1=-D1
220 IF Y<60 OR Y>135 THEN D2=-D2
230 X=X+D1 : Y=Y+D2
240 SS=PEEK(V+30) : SB=PEEK(V+31)
250 IF SS=0 AND SB=0 THEN 180
260 POKE S+4,17 : PRINT"*" : GOTO 180
270 DATA 3,231,192,15,0,240
280 DATA 30,102,120,31,195,248
290 DATA 63,0,252,62,126,124
300 DATA 108,153,54,97,255,134
310 DATA 99,255,198,65,255,236
320 DATA 31,255,248,15,255,240
330 DATA 63,255,252,127,255,254
340 DATA 143,255,241,31,255,248
350 DATA 103,255,230,139,255,209
360 DATA 50,126,76,34,0,68
370 DATA 3,0,192
READY.
```

Every time the two sprites collide, the computer PRINTs an asterisk. Eventually the asterisks will scroll up the screen, creating a continuous "collision":

SPRITE COLLISION DISPLAYS





PIE CHARTS AND GRAPHS

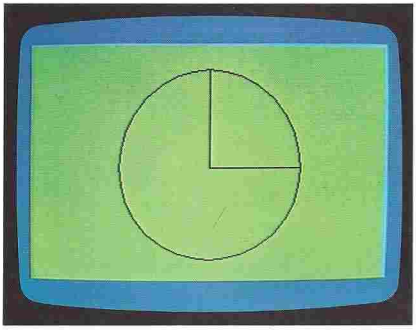
Computer graphics are ideal for displaying information that you can take in at a glance, and one of the most easily understood displays that the Commodore can produce is a pie chart. Pie charts are particularly good for showing the relative sizes of numbers, or how different items make up parts of a whole.

Starting off a pie chart program

To produce a pie chart, the first job is to draw a circle. You can then put in the edges that mark off the "slices". In the following program, one right-angled slice is drawn in a circle. To make the Commodore produce a circle you need to use high resolution, so before you RUN this program remember that you need to move the BASIC area (see page 14) and then LOAD the complete set of eight graphics subroutines, which you should have on tape or disk. Here's the simple pie chart program that calls these routines:

FIXED PIE CHART PROGRAM

```
LIST
10 GOTO 1000
1000 POKE 53280,6 : COL=5
1010 GOSUB 100 : GOSUB 200
1020 XC=160 : YC=100
1030 RAD=80 : GOSUB 700
1040 LX=240 : LY=100
1050 MX=160 : MY=100 : GOSUB 600
1060 MX=160 : MY=20 : GOSUB 600
1070 GOTO 1070
READY.
```



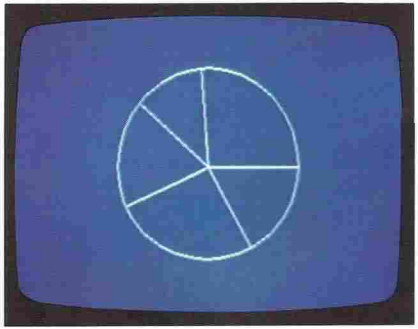
The program draws a circle at the center of the screen (co-ordinates 160,100). Line 1050 then draws the first radius from the right to the center. Line 1060 then draws the second radius straight up to form the slice.

Adding more slices to the chart

You could go on from this to add lines to draw further slices and so build up the chart, but this sort of program wouldn't really be much use. A fixed pie chart program can only ever give you the same display. What is much more useful is a program which responds to the numbers you key in:

VARIABLE PIE CHART PROGRAM

```
LIST
10 GOTO 1000
1000 PRINT CHR$(147) : T=0
1010 INPUT "HOW MANY SLICES";N : PRINT
1020 DIM P(N) : FOR C=1 TO N
1030 PRINT "SIZE OF SLICE";C;" ";
1040 INPUT S : T=T+S : P(C)=T : NEXT C
1050 P(0)=0 : POKE 53280,6 : COL=22
1060 GOSUB 100 : GOSUB 200
1070 XC=160 : YC=100
1080 RAD=80 : GOSUB 700
1090 FOR C=0 TO N-1
1100 LX=160 : LY=100 : S=P(C)*2*π/T
1110 MX=INT(160+(80*COS(S)+0.5))
1120 MY=INT(100+(80*SIN(S)+0.5))
1130 GOSUB 600 : NEXT C
1140 GOTO 1140
READY.
```



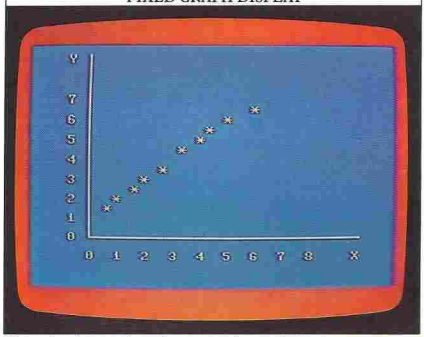
In this program lines 1000 to 1040 sort out how many slices need to be drawn on the pie chart and how big each slice needs to be. You can specify all these values. In line 1020 you will see the command DIM. This sets up something called an array, which is a way of holding

and retrieving information easily. Arrays are explained in full on pages 52–53. From the information held in the array, the program calculates the angle of each slice. This calculation and the actual drawing are carried out by lines 1070 to 1130 (because the chart is shown in high resolution, you can't PRINT any labels on it).

Putting information into graphs

Pie charts are useful for showing how something is split up; graphs, on the other hand, show how two separate sets of data are related. Here's a simple Commodore graph display. It's produced in low (text) resolution, so that unlike the pie chart display, it can be labeled, and it is also created more quickly:

FIXED GRAPH DISPLAY



You don't need to be a mathematician to get some useful information from this graph. As time goes by (along the horizontal axis), the amount measured by the vertical axis is steadily increasing.

The program that sets up this graph has to draw the axes, label them and PRINT the asterisks:

FIXED GRAPH PROGRAM

```
LIST

10 PRINT CHR$(147)
20 POKE 53280,2 : POKE 53281,6
30 PRINT CHR$(158) : FOR Y=0 TO 17
40 POKE 214,Y : PRINT : POKE 211,6
50 PRINT CHR$(225) : NEXT Y
60 PRINT TAB(6);CHR$(173);
70 FOR X=7 TO 35
80 PRINT TAB(X);CHR$(96);: NEXT X
90 PRINT : PRINT : PRINT "  0  1  2
 3  4  5  6  7  8  X" : PRINT
100 C=0
110 FOR Y=18 TO 4 STEP -2
120 POKE 214,Y : PRINT : POKE 211,3
130 PRINT C; : C=C+1 : NEXT Y
140 POKE 214,0 : PRINT : PRINT "    Y"
150 FOR D=1 TO 10 : READ X,Y
160 POKE 214,Y : PRINT : POKE 211,X
170 PRINT "*" : NEXT D
180 GOTO 180
190 DATA 8,15,9,14,11,13,12,12,14,11
200 DATA 16,9,18,8,18,7,21,6,24,5
READY.
```

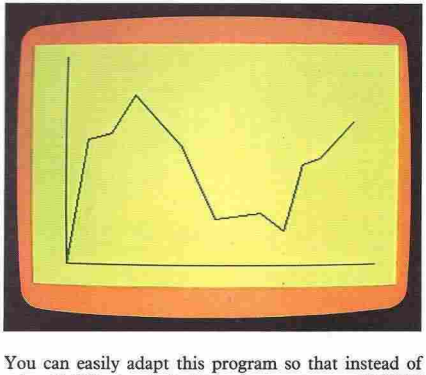
The two sets of information are contained in the DATA in lines 190 and 200. Lines 30 to 140 draw and label the two axes of the graph. The loop that follows READs 10 pairs of values from the DATA statements and uses these as co-ordinates for plotting points.

Programming a high-resolution graph plotter

The fixed graph display is quite coarse, because it uses the low-resolution screen which only gives 504 character positions within the graph's area of 28 columns by 18 rows. For a more detailed graph, you need to go to high resolution (using the high-resolution subroutines). The next program does this, again working with information held as DATA. The program takes longer to RUN, but the display is more precise:

HIGH-RESOLUTION GRAPH PROGRAM

```
LIST
10 GOTO 1000
1000 COL=7 : POKE 53280,8
1010 GOSUB 100 : GOSUB 200
1020 LX=30 : LY=10
1030 MX=30 : MY=180 : GOSUB 600
1040 MX=300 : MY=180 : GOSUB 600
1050 LX=30 : LY=180
1060 FOR C=1 TO 10 : READ MX,MY
1070 GOSUB 600 : NEXT C
1080 GOTO 1080
1090 DATA 50,75,70,70,90,40,130,80
1100 DATA 160,140,200,135,220,150
1110 DATA 235,95,250,90,280,60
READY.
```



You can easily adapt this program so that instead of using DATA already built in, it accepts your INPUT values before drawing the graph – just like the variable pie chart program – so that you can see information displayed as a graph in a few seconds.

BAR CHARTS

Having seen how your Commodore can produce high-resolution pie charts and graphs, you can now add a third way of showing information graphically, by using a low-resolution bar chart.

In bar charts data is displayed not as single points, but as columns whose height depends on the size or level of the item shown. They are frequently used to show changes in currency values, votes in elections and so on, and you can easily make your Commodore use them to show personal data in an instant graphic way.

Writing a bar chart program

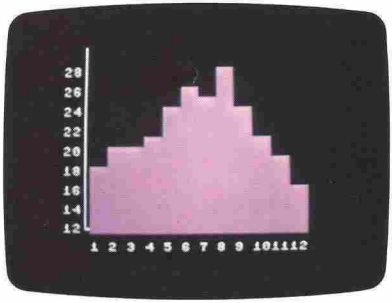
Because a bar chart is really an adapted graph, you can use much the same programming techniques to produce one. The main difference is that instead of plotting a single point when fed with co-ordinates, the program must PRINT a column. With the Commodore, columns are most easily made up with the graphics square, using PRINT CHR$(18), and you can then add some color for clarity. The next program does this by using information which you can INPUT. Because it uses the low-resolution screen, the chart can also be labeled without any problem:

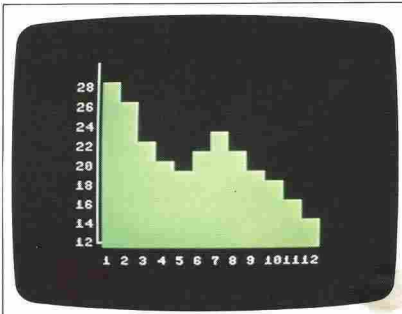
SIMPLE BAR CHART PROGRAM

```
LIST
 10 PRINT CHR$(147);CHR$(5)
 20 POKE 53280,0 : POKE 53281,0
 30 FOR Y=0 TO 17
 40 POKE 214,Y : PRINT : POKE 211,6
 50 PRINT CHR$(125); : NEXT Y
 60 PRINT TAB(6);CHR$(173)
 70 PRINT : PRINT "          1 2 3 4 5 6 7 8
    9 101112"
 80 C=12 : FOR Y=18 TO 2 STEP -2
 90 POKE 214,Y : PRINT : POKE 211,3
100 PRINT C; : C=C+2 : NEXT Y
110 PRINT CHR$(156) : FOR M=1 TO 12
120 POKE 214,22 : PRINT : POKE 211,12
130 PRINT "X";M;"= "; : INPUT T
140 POKE 214,22 : PRINT : POKE 211,12
150 PRINT "
160 FOR R=18 TO 30-T STEP -1
170 POKE 214,R : PRINT : POKE 211,M*2+5
180 PRINT CHR$(18);"  " : NEXT R
190 NEXT M
200 GOTO 200
READY.
```

The Y (vertical) axis is drawn by lines 30 to 50 in the same position as for the first graph on page 35. The PRINT statement at line 70 labels the X (horizontal) axis with the numbers 1 to 12, which could represent the months of the year. The program produces bars that are two graphics squares wide. If you want to increase the number of bars that you can show on each chart, you can reduce the width of each one to a single square. But if you do this, remember that you will also have to alter the PRINTing positions to make the bars appear the right distance apart. Here is the double-square program in action, showing it in two different colors:

SIMPLE BAR CHART DISPLAYS





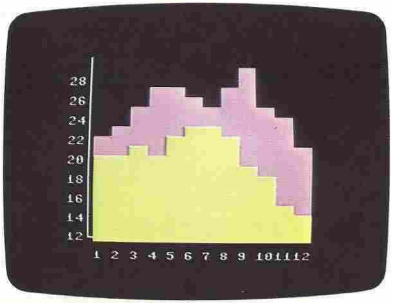
Adding charts together

The first charts can show just one list of items. But it is possible to reorganize them so that they can display more than one set of information. You may for instance want to see both maximum and minimum figures like temperatures on the same chart. You don't have to rewrite the first program. A few additions will do:

```
105 FOR N=1 TO 2
115 IF N=2 THEN PRINT CHR$(158)
195 NEXT N
```

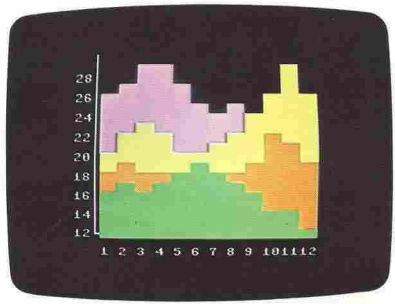
This RUNs as before until you have finished keying in the first set of data. It then sets N to 2 and PRINTs columns of yellow squares instead of magenta ones. Your second set of data must be generally smaller than the first set, otherwise the magenta chart will be completely overwritten by the yellow one:

DOUBLE COLOR CHART



You can extend this to any number of overlapping charts by increasing the upper limit of the FOR ... NEXT loop in line 105, and then by adding extra lines of program between lines 110 and 120 to change color. Here's a chart which shows four sets of information:

4-COLOR CHART



Improving your charts with alternating color

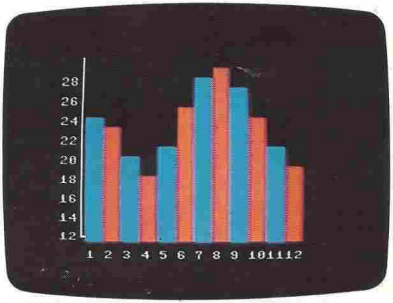
One of the problems with charts that have a single color for each set of data is that you cannot distinguish individual bars, making it difficult to relate each bar height to scale on the bottom axis. You can get around this by using two different colors again, but this time by alternating them as the bars are PRINTed. It's then quite easy to see which bar relates to which figure on the X axis.

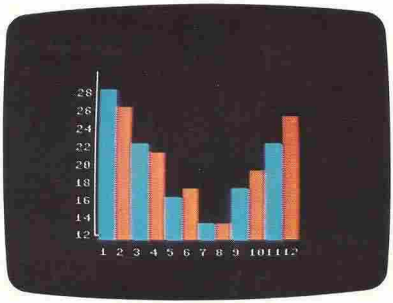
The following program is an adaptation of the simple bar chart. If you take out the lines that make it show more than one set of data, you can then edit it to produce a display with alternating colors.

Instead of having the color fixed, it's now controlled by the variable A. A loop is used in conjunction with IF ... THEN to set the drawing color to either blue or red. You can use this type of color-changing loop with as many colors as you like. Here is the program and some of the displays that it can produce:

ALTERNATING COLORS CHART

```
10 PRINT CHR$(147);CHR$(5)
20 POKE 53280,0 : POKE 53281,0
30 FOR Y=0 TO 17
40 POKE 214,Y : PRINT : POKE 211,6
50 PRINT CHR$(125) : NEXT Y
60 PRINT TAB(6);CHR$(173)
70 PRINT " 1 2 3 4 5 6 7 8
 9 10 11 12"
80 C=12 : FOR Y=18 TO 2 STEP -2
90 POKE 214,Y : PRINT : POKE 211,3
100 PRINT C ; : C=C+2 : NEXT Y : M=1
110 FOR A=1 TO 2 : PRINT CHR$(5)
120 POKE 214,21 : PRINT : POKE 211,12
130 PRINT "X";M;"=";A : INPUT T
140 POKE 214,21 : PRINT : POKE 211,12
150 PRINT
160 PRINT CHR$(34-3*A)
170 FOR R=18 TO 30-T STEP -1
180 POKE 214,R : PRINT : POKE 211,M*2+5
190 PRINT CHR$(18);" " : NEXT R
200 M=M+1 : NEXT A : IF M<12 THEN 110
210 GOTO 210
READY.
```





GRAPHICS WITH GRAVITY

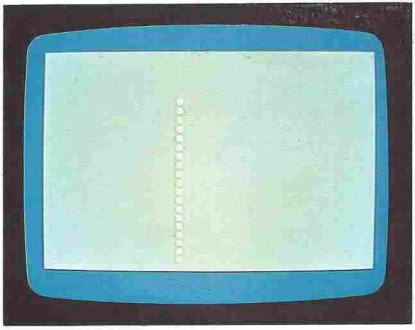
On pages 24–25 you saw how the Commodore can produce "natural" graphics, shapes that you can sometimes see in the natural world. To make these shapes you can simply experiment with the graphics subroutines and see what happens. But if you want the computer to simulate something moving in a realistic way, an understanding of how it moves in real life will help you a great deal in programming the same movement on the computer screen.

How the Commodore starts a ball falling

On pages 8–9 IF ... THEN was used to "bounce" a ball in straight lines moving at a constant speed. However, a ball doesn't move in straight lines. On the screen below is a short program to demonstrate how you could begin simulating a more realistic fall (the display beneath it includes after-images normally deleted by the first statement in line 80):

SIMPLE FALL PROGRAM

```
LIST

10 PRINT CHR$(147);CHR$(158)
20 POKE 53280,6 : POKE 53281,12
30 R=5 : C=16 : V=1
40 POKE 214,R : PRINT : POKE 211,C
50 PRINT "●";
60 FOR T=1 TO 1500 : NEXT T
70 POKE 214,R : PRINT : POKE 211,C
80 PRINT "^^" : R=R+V : IF R>22 THEN END
90 POKE 214,R : PRINT : POKE 211,C
100 PRINT "●"
110 FOR T=1 TO 50 : NEXT T
120 GOTO 70
READY.
```



Falling objects are influenced by several forces – gravity, air resistance, surface friction and something called the "coefficient of restitution" – which make them move in a complex way. However, you don't have to be a physicist to write a more realistic program than this. If you drop a ball, it falls to the ground and bounces up again, and that's all you need to know to get a ball bouncing on the screen.

Programming movement in two directions

In the simple fall program, line 50 PRINTs the ball near the top of the screen. After a 2-second pause, the ball starts to move downward. Line 80 erases it, the row number is then increased by 1 and last the ball is PRINTed again. If you RUN this program, you will find that although the ball is indeed falling to the bottom of the screen, its movement doesn't look very realistic. The program also ends abruptly when the ball reaches the bottom of the screen. The next program improves the display considerably by making the ball move sideways as well:

SIDEWAYS FALL PROGRAM

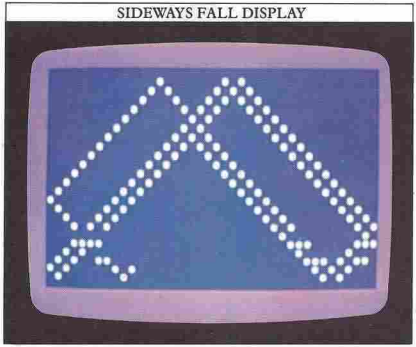
```
LIST

10 PRINT CHR$(147);CHR$(5)
20 POKE 53280,4 : POKE 53281,6
30 R=5 : C=16 : V=1 : H=1
40 POKE 214,R : PRINT : POKE 211,C
50 PRINT "●";
60 FOR T=1 TO 1500 : NEXT T
70 POKE 214,R : PRINT : POKE 211,C
80 PRINT "^^" : R=R+V : C=C+H
85 IF R>22 THEN END
90 POKE 214,R : PRINT : POKE 211,C
100 PRINT "●"
110 FOR T=1 TO 50 : NEXT T
120 GOTO 70
READY.
```

The variable H represents the change in horizontal position and V the change in vertical position. On each loop, V is added to the row number and H to the column number. Now it's easy to modify the motion in any direction. For instance, you can make the ball bounce by adding:

```
85 IF C<1 OR C>38 THEN H=-H
86 IF R<1 OR R>22 THEN V=-V
```

You have seen these techniques using AND and OR with IF ... THEN before, so these two lines should present you with no problems. If you take out the lines which erase the ball as it moves, you will now see a display like this:

SIDEWAYS FALL DISPLAY



Computer-controlled gravity

Although the ball bounces around the screen it doesn't yet look completely realistic. The reason for this is that there is no gravity acting on it. You can add a "force" like gravity that pulls in any direction, or that even changes direction during a program's RUN. Gravity acts downward, so, as the ball moves from the top to the bottom of the screen it should accelerate. When it bounces back up, it should slow down until it falls back again. The next program imitates this:

BOUNCING BALL PROGRAM

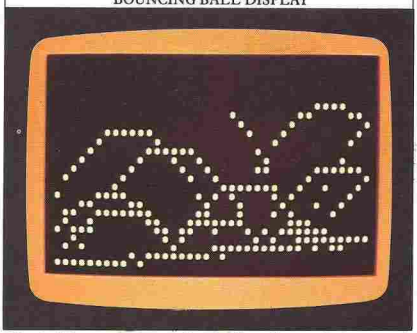
```
LIST

 10 PRINT CHR$(147);CHR$(158)
 20 POKE 53280,0 : POKE 53281,0
 30 S=54272 : POKE S+24,15
 40 POKE S+5,0 : POKE S+6,240
 50 POKE S,0 : POKE S+1,60
 60 R=5 : C=22 : H=1 : V=1
 70 POKE 214,R : PRINT : POKE 211,C
 80 PRINT CHR$(113);
 90 FOR T=1 TO 20 : NEXT T
100 POKE 214,R : PRINT : POKE 211,C
110 PRINT " ";
120 V=V+0.2 : R=R+V : C=C+H
125 IF R>23 THEN R=25
130 IF C>1 AND C<38 THEN 160
140 POKE S+4,33 : FOR T=1 TO 20
150 NEXT T : POKE S+4,32 : H=-H
160 IF R<20 THEN 70
170 POKE S+4,33 : V=-V : FOR T=1 TO 20
180 NEXT T : POKE S+4,32 : GOTO 70
READY.
```

In this program, the gravity factor is written in at line 120. The addition of 0.2 to V means that the change in R – the vertical position – is no longer constant. It increases on each loop, speeding the ball up.

When the ball hits the bottom of the screen a sound is produced, and the ball's direction is reversed by line 170. V then becomes negative, repeatedly decreasing the row number. The added gravity factor at line 120 makes V less and less negative, slowing down the upward progress of the ball until its vertical movement ceases, V becomes positive again, and the ball begins to move downward once more.

This display shows how the ball moves with this program (again, this is what you will see if you stop the computer deleting the after-images by masking line 110 with a REM command):

BOUNCING BALL DISPLAY



The ball bounces around as before, but as it does so, it doesn't reach the same height on each bounce. Its height is gradually decreasing, although its horizontal movement remains the same. The result of this is a rough example of a curve known as a parabola. Eventually the ball will reach the bottom of the screen when the program goes into an endless loop.

In just the same way as you can influence vertical movement by "gravity", you can alter the horizontal movement as well. This gives the impression of an object that is not only falling under gravity, but which is also being blown along by a strong wind.

Simulating gravity with high resolution

The curve that the ball makes in the gravity program isn't very smooth because the ball is a text character, and there are only 40×25 possible positions that it can be shown at. If you want to produce smoother bouncing, you can experiment with plotting high-resolution ball tracks instead. This will produce a single point at a graphics co-ordinate pair, allowing much smoother movement curving over a 320×200 high-resolution display.

To do this, however, you would have to modify the program so that the low-resolution co-ordinates in all the lines were converted to high-resolution co-ordinates. If you refer to the grid on page 58, you shouldn't find this too difficult. Being in high resolution, the program will work more slowly than the original one, although the curves produced will be more realistic.

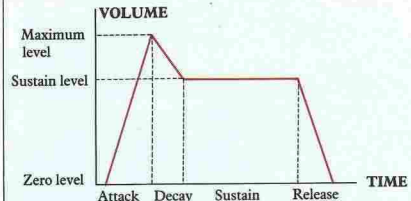
SHAPING SOUND

As you may have seen in Book 1, the Commodore has a sound facility that is unusually powerful, allowing you to produce a wide range of notes and sound effects. The Sound Interface Device, or SID chip, can do a lot more than just producing a sound at a particular frequency. Here you will find out more about how to control the profile of a sound, a characteristic that is known as the sound "envelope" or ADSR.

How a sound changes during playing

The envelope shape of a sound or note is a graph which shows how the volume changes as the sound progresses from start to finish. On the Commodore there are four parameters that can be varied to shape volume. These are called Attack, Decay, Sustain and Release (ADSR).

THE "SHAPE" OF A TYPICAL SOUND



Attack is the time taken from the start for the volume to reach its maximum. This can be anything from 2 milliseconds to 8 seconds. Decay is the part of the sound from the end of the attack period to the time that the sound reaches its sustain level. The decay time can be between 6 milliseconds and 24 seconds. Sustain is different from the other three parameters in that it is not a time but a volume level, expressed as a fraction of the maximum value reached at the end of the attack phase. A value of 0 indicates that the sound will decay to a sustain level of 0, and a value of 15 indicates that the sustain value will be the same as the maximum level, so that the sound does not decay at all. The sustain level is maintained by the SID chip until the sound is switched off. Release is the length of time taken for the volume to decay from its sustain value to nothing, and this can range from 6 milliseconds to 24 seconds. To program all these, you need to select the appropriate ADSR settings.

Here is a program which demonstrates the effect that the ADSR settings have. It plays a simple tune, changing the ADSR settings each time:

ADSR RANGES

This table shows the effect of all the settings from 0–15 on each of the ADSR parameters.

Setting	Attack time (sec.)	Decay time (sec.)	Sustain level (%)	Release time (sec.)
0	0.002	0.006	0	0.006
1	0.008	0.024	7	0.024
2	0.016	0.048	13	0.048
3	0.024	0.072	20	0.072
4	0.038	0.114	27	0.114
5	0.056	0.168	33	0.168
6	0.068	0.204	40	0.204
7	0.08	0.24	47	0.24
8	0.1	0.3	53	0.3
9	0.25	0.75	60	0.75
10	0.5	1.5	67	1.5
11	0.8	2.4	73	2.4
12	1	3	80	3
13	3	9	87	9
14	5	15	93	15
15	8	24	100	24

ADSR DEMONSTRATION PROGRAM

```
LIST -180

10 PRINT CHR$(147)
20 S=54272
30 DIM N(11,2) : POKE S+3,8
40 RESTORE : FOR C=0 TO 11
50 READ N(C,0)
60 READ N(C,1)
70 READ N(C,2)
80 NEXT C : POKE S+24,15
90 FOR C=1 TO 4
100 READ W
110 READ AD
120 READ SR
130 POKE S+5,AD
140 POKE S+6,SR
150 FOR K=0 TO 11
160 POKE S,N(K,1)
170 POKE S+1,N(K,0)
180 POKE S+4,W+1

READY.
```

```
LIST 190-

190 FOR T=1 TO N(K,2) : NEXT T
200 POKE S+4,W
210 NEXT K
220 FOR T=1 TO 1000 : NEXT T
230 NEXT C
240 DATA 25,29,160,25,29,720
250 DATA 21,30,160,22,96,480
260 DATA 25,29,160,28,49,160
270 DATA 31,164,80,33,134,160
280 DATA 28,49,80,31,164,160
290 DATA 25,29,160,22,96,960
300 DATA 64,9,0
310 DATA 32,144,243
320 DATA 16,9,163
330 DATA 64,88,57
READY.
```

Programming a sound profile

Each of the three sound channels on the Commodore has two registers associated with it to control ADSR. Attack and decay are together in one SID chip register, and sustain and release are together in the other.

ADSR REGISTERS

Each register is made up from two half-bytes or nibbles. These control separate features of the sound.

SID register (S=54272)	Sound channel	ADSR function high nibble	low nibble
S+5	1	Attack	Decay
S+6	1	Sustain	Release
S+12	2	Attack	Decay
S+13	2	Sustain	Release
S+19	3	Attack	Decay
S+20	3	Sustain	Release

The SID chip registers are split into two equal parts, each four bits long. Attack, decay, sustain and release are all controlled by one of these "nibbles". Having four bits gives a total of 16 possible settings. Decay and release are set by the low nibbles. So a setting of 10, for example, means a nibble value of 10. Attack and sustain however are set by the high nibbles. For them, a value of 10 would mean a nibble value of 160.

ADSR VALUES

Decay and release are controlled by low-value nibbles, so the setting number and nibble values have the same range (0–15). Attack and sustain are controlled by high-value nibbles and so have to be converted from the setting numbers between 0 and 15.

Setting	Attack/Sustain value	Decay/Release value
0	0	0
1	16	1
2	32	2
3	48	3
4	64	4
5	80	5
6	96	6
7	112	7
8	128	8
9	144	9
10	160	10
11	176	11
12	192	12
13	208	13
14	224	14
15	140	15

Suppose that you wanted to program a sound which had an attack parameter setting of 6 and a decay setting of 11, using sound channel 2. From the table above you can see that an attack setting of 6 has a value of 96, and a decay setting of 11 has a value of 11.

HOW ONE BYTE CONTROLS TWO PARAMETERS

By adding together the values for two nibbles in a specific byte, you can make one ADSR byte control two separate features of a sound.

	ATTACK nibble				DECAY nibble			
Bit values	128	64	32	16	8	4	2	1
	0	1	1	0	1	0	1	1

ATTACK nibble value=96 DECAY nibble value=11

Total ATTACK/DECAY byte value=107

Adding these values together gives 107 which is the value to POKE into SID register S+12 – the register which controls attack and decay on sound channel 2.

Here's a program which shows you the variety of sounds these settings can produce. It allows you to control waveform and ADSR as a tune is played:

USER-CONTROLLED ADSR PROGRAM

```
LIST -180

10 S=54272
20 DIM N(11,2) : POKE S+3,8
30 RESTORE : FOR C=0 TO 11
40 READ N(C,0)
50 READ N(C,1)
60 READ N(C,2)
70 NEXT C : POKE S+24,15
80 PRINT CHR$(147)
90 PRINT "WHICH WAVEFORM"
100 PRINT "1=TRIANGULAR 3=PULSE"
110 PRINT "2=SAWTOOTH   4=RANDOM NOISE"
120 INPUT "ENTER 1 TO 4 ";W
130 INPUT "ATTACK RATE (0-15)";A
140 INPUT "DECAY RATE (0-15)";D
150 INPUT "SUSTAIN LEVEL (0-15)";L
160 INPUT "RELEASE RATE (0-15)";R
170 AD=16*A+D : SR=16*L+R
180 POKE S+5,AD : POKE S+6,SR

READY.
```

```
LIST 190-

190 FOR K=0 TO 11
200 POKE S,N(K,1)
210 POKE S+1,N(K,0)
220 POKE S+4,2+(W*2)+1
230 FOR T=1 TO N(K,2) : NEXT T
240 POKE S+4,2+(W*2)
250 NEXT K
260 GOTO 80
270 DATA 25,29,160,25,29,720
280 DATA 25,30,160,22,96,90
290 DATA 25,29,160,28,49,160
300 DATA 31,164,80,33,134,160
310 DATA 28,49,80,31,164,160
320 DATA 25,29,160,22,96,960
READY.
```

ADVANCED SOUND EFFECTS

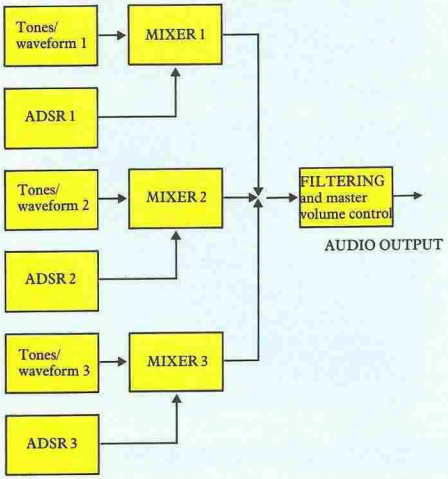
You are now ready to take a look at some of the Commodore's more advanced sound facilities, including filtering and ring modulation. These techniques can be quite tricky to master, and you can only become fully familiar with them through many hours of practice. The details on these two pages will point you in the right direction, and from there you can make your own way by experimentation. With all the facilities of the SID chip behind you, this can lead to many hours of discovery.

Filtering a sound effect

Except for the purest sound tones (called sine wave sounds), all sounds are made up of many frequencies. One of these frequencies, called the fundamental, is the dominant one which gives the sound its pitch. The others are multiples of this frequency and are called harmonics. The second harmonic has twice the frequency of the fundamental, the third harmonic three times the frequency and so on.

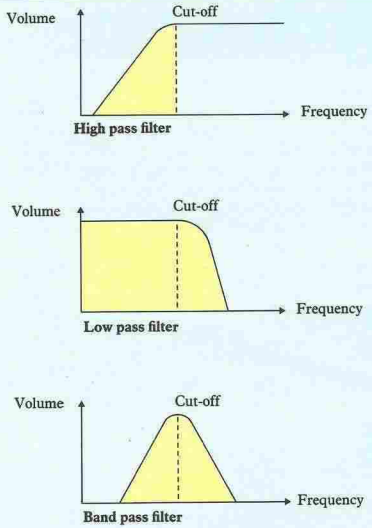
The characteristics of a sound can be changed dramatically by altering the volume levels of just a few of the harmonics in the sound, and within the SID chip this job can be done by filters. The output from each channel is first set up by the tone and waveform settings. This output is then modified by the ADSR which is brought in at the mixing stage. The outputs from all three channels are then put through a filtering and master volume control to give the final sound.

HOW SOUND IS MIXED AND FILTERED



As you can see from this, any filtering that is done simultaneously affects the output of all three channels. There are three different types of filter available within the SID chip and they can be used either individually or in combination. They are known as low pass, band pass and high pass filters. Each of the filters has a "cut-off" frequency, which is the point in the frequency range where the filter starts to become effective cutting out parts of the sound.

SOUND FILTER PROFILES



The filters themselves are controlled by bits stored in four registers within the SID chip, S+21 to S+24. S+21 and S+22 contain an 11-bit number, three bits in location S+21 and eight in location S+22, which controls the cut-off frequency in the range 30Hz to 12kHz. S+23 controls which sound channels will be filtered (low 3 bits) and how steeply the filters will be cut off (high 4 bits), and S+24 controls which filters will be switched on and also the master volume level.

The next programs generate white noise from a sound channel and then apply the band pass filter to this output, changing the cut-off frequency so that you can hear the effect on the sound:

ROCKET PROGRAM

```
LIST

10 PRINT CHR$(147)
20 S=54272
30 POKE S+5,0
40 POKE S+6,240
50 POKE S,0
60 POKE S+1,8
70 POKE S+4,129
80 FOR T=1 TO 500 : NEXT T
90 POKE S+23,241
100 POKE S+24,47
110 FOR C=1 TO 200
120 POKE S+22,C
130 FOR T=1 TO 50 : NEXT T
140 NEXT C
150 POKE S+4,128
READY.
```

WEAPON FIRE PROGRAM

```
LIST

10 PRINT CHR$(147)
20 S=54272
30 POKE S+5,0
40 POKE S+6,240
50 POKE S,0
60 POKE S+1,20
70 POKE S+4,33
80 POKE S+23,241
90 POKE S+24,47
100 FOR K=1 TO 3
110 FOR C=0 TO 160 STEP 20
120 POKE S+22,C
130 NEXT C
140 FOR C=160 TO 0 STEP -2
150 POKE S+22,C
160 NEXT C
170 NEXT K
180 POKE S+4,16
190 FOR T=1 TO 1000:NEXT
200 GOTO 10
READY.
```

Changing sounds with ring modulation

Ring modulation is a process by which the triangular waveform output of a selected sound channel is replaced by a modulated combination of it and the output from the next channel. So, for example, selecting ring modulation on channel 1 will replace the channel 1 triangular output by a ring modulated combination of channels 1 and 2. Ring modulation on channel 2 works on channels 2 and 3.

You can select this effect by setting the third bit in the channel's control register. To do this, you need to use bit masking. The third bit has a decimal value of 4, so to turn it on, you will need to POKE a value using OR 4

RING MODULATION REGISTERS

Channel	Control register
1	2+4
2	S+11
3	S+18

so that only this bit is set. This line will do it:

POKE S+CR,PEEK(S+CR) OR 4

Here CR is the SID chip control register number for the required channel. AND will turn the same bit off:

POKE S+CR,PEEK(S+CR) AND 251

Here are two programs which let you hear ring modulation at work. The first produces the sound of a bell, and the second the sound of an alarm:

RINGING BELL PROGRAM

```
LIST

10 PRINT CHR$(147)
20 S=54272
30 POKE S+24,15
40 POKE S,0
50 POKE S+1,158
60 POKE S+5,9
70 POKE S+6,0
80 POKE S+15,30
90 POKE S+4,21
100 FOR T=1 TO 200 : NEXT T
110 POKE S+4,20
120 GOTO 30
READY.
```

ALARM PROGRAM

```
LIST

10 PRINT CHR$(147)
20 S=54272 : D=150
30 POKE S+24,15
40 POKE S,0
50 POKE S+1,D
60 POKE S+5,6
70 POKE S+6,0
80 POKE S+15,30
85 FOR K=1 TO 50
90 POKE S+4,21
100 FOR T=1 TO 40+K/1.5 : NEXT T
110 POKE S+4,20
120 POKE S+1,210-D
130 POKE S+4,21
140 FOR T=1 TO 40+K/1.5 : NEXT T
150 POKE S+4,20
160 NEXT K
READY.
```

Watch out for this simple bug

When you're working with sound programs, you may find that some of them just don't seem to work, even though the listings look perfect. Your problem here might be that a previous sound program has POKEd register(s) in the SID chip that are interfering with your new program. If this does happen, the best thing to do is briefly switch off the computer to clear out the SID chip, and then key the new program in again.

WORKING WITH WORDS

Until now, you have treated strings – or words that make up strings – as indivisible units. Some of the programs so far have added strings together, but none of them have "looked inside" the quotation marks that begin and end every string to work on the characters that are there. With the Commodore you can take strings apart and reassemble their characters in a number of different ways. This means that you can program the computer to take out part of a word or group of words and examine it – a process that can be very useful.

Like most computers that work with BASIC, the Commodore has a family of commands that can be used to manipulate strings. Some of the most useful are LEFT$, RIGHT$ and MID$. They are used to pick out the first, last or middle character of a string respectively.

How to cut up words

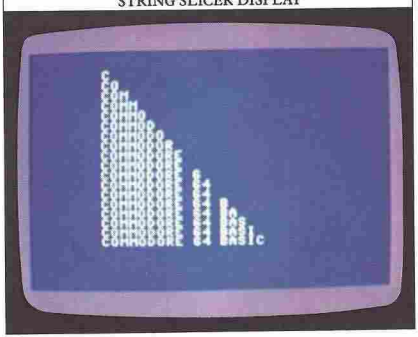
You can make the Commodore break into a word by slicing parts off the string. It's fairly straightforward. To see how to do it, first type in this program:

STRING SLICER PROGRAM

```
LIST

10 POKE 53280,4:POKE 53281,6
20 A$="COMMODORE 64 BASIC"
30 PRINT CHR$(147) : PRINT
40 FOR N=1 TO 18
50 PRINT TAB(8);LEFT$(A$,N)
60 NEXT N
READY.
```

The special technique here is in line 50, where a LEFT$ command appears as part of a string expression. For each value of N, line 50 PRINTs a string N characters long, from the first character, C, to the Nth character. So, the first line contains the string "C", the second line "CO" and so on, until N equals the length of the string that is set. With this program you can use any string – a group of words, numbers or other symbols; it's best if the value of N is not more than one screen line (40 characters). If you use a different string, make sure that the maximum value of N in line 40 is the same as the length of your string. Here's the display that the string slicer produces with line 20 set as above:

STRING SLICER DISPLAY



You can use this kind of technique to pick out strings that all begin with the same letter or word, and then perhaps PRINT them out in a series of lists.

An opposite effect is just as easy to produce. Try adding this to the first program:

```
70 FOR N=1 TO 18
80 PRINT TAB(8); RIGHT$(A$,19-N)
90 NEXT N
```

Now, as N increases from 1 to 18, the length of the string PRINTed decreases from 18 characters to only 1, as letters are sliced away from the left.

Picking out parts of a phrase

Now you can explore this technique. The next program shows how the Commodore can select parts of a string and use them in different ways:

SELECTIVE STRING SLICER

```
LIST

10 A$="DATA 1/DATA 2/DATA 3"
20 PRINT CHR$(147):POKE 53280,8:POKE 532
81,11
30 POKE 214,7 : PRINT : POKE 211,6
40 PRINT A$
50 POKE 214,10 : PRINT : POKE 211,6
60 PRINT LEFT$(A$,6)
70 PRINT TAB(6);"----------"
80 POKE 214,12 : PRINT : POKE 211,6
90 PRINT MID$(A$,8,6)
100 PRINT TAB(6);"----------"
110 POKE 214,14 : PRINT : POKE 211,6
120 PRINT RIGHT$(A$,6)
130 PRINT TAB(6);"----------"
READY.
```

As you can see from the display this produces, you aren't limited to dealing with the first N characters of a string. In fact, you can take any consecutive group of characters from a word or sentence. In this program line 60 works in the same way as line 50 of the first slicing program. Line 90 forms a string of 6 characters from characters 8 to 14 out of the middle of A$. Finally, line 120 forms a third string from the last six characters. Although these three "substrings" are formed from parts of A$, A$ itself is still intact. This lets you take a group of words and pick out any of them for use on their own in a program.

Word games with string commands

The next program shows how you can use these methods of handling words in a game. It's a computerized "hangman" word-guessing contest in which one player enters a word and the other has to guess it; the computer PRINTs letters guessed correctly in their right positions in the word, and also lets you try guessing the whole thing:

WORD GAME PROGRAM

```
LIST -110

10 POKE 53280,0 : POKE 53281,0 : PRINT C
HR$(147) : PRINT TAB(14);"HANGMAN"
20 POKE 214,8 : PRINT : PRINT TAB(5);"AS
K A FRIEND TO TYPE A WORD"
30 POKE 214,10 : PRINT : PRINT TAB(5);"O
R PHRASE, FOR YOU TO GUESS"
40 POKE 214,15 : PRINT : PRINT "    **** DO
N'T LOOK AT THE SCREEN! ****"
50 PRINT : PRINT "  PRESS RETURN WHEN YO
U'VE FINISHED"
60 PRINT TAB(10); : INPUT "WORD = ";A$
70 PRINT CHR$(147) : PRINT TAB(2); "HANG
MAN - PRESS 1 TO MAKE A GUESS"
80 P=(32-LEN(A$))/2 : S=0 : POKE 214,7 :
 PRINT
90 FOR N=1 TO LEN(A$) : POKE 211,P+N
100 IF MID$(A$,N,1)=" " THEN PRINT " ";
 : NEXT N
110 PRINT "-"; : NEXT N

READY.
```

```
LIST 120-

120 POKE 214,18 : PRINT : PRINT TAB(9);
: INPUT "TRY A LETTER";T$
130 IF T$="1" THEN 180
140 S=S+1 : POKE 214,13 : PRINT : PRINT
TAB(2);"TRIES = ";S : POKE 214,7
150 PRINT : FOR N=1 TO LEN(A$)
160 IF T$=MID$(A$,N,1) THEN POKE 211,P+N
 : PRINT T$;
170 NEXT N : GOTO 120
180 POKE 214,18 : PRINT : PRINT TAB(7);
: INPUT "ENTER YOUR GUESS";T$
190 PRINT CHR$(147)
200 POKE 214,10 : PRINT : POKE 211,14
210 IF T$=A$ THEN PRINT "CORRECT"
220 IF T$<>A$ THEN PRINT "-WRONG-"
230 FOR T=1 TO 5000 : NEXT T : GOTO 10
READY.
```

Lines 10 to 60 PRINT the title frame. When a friend has typed in the test string that you will have to guess, line 80 calculates the length of this test string using the command LEN, and sets the score (S) to zero.

The program now has to PRINT symbols on the screen to represent the letters in the test string. As you guess the letters, any correctly guessed letters will replace these symbols. Also, to allow for test phrases rather than just words, the positions of the spaces between the words are shown. Line 110 PRINTs hyphens to represent the characters. Line 90 uses the value of P to work out where the characters that represent the test string should be PRINTed so that they lie in the middle of the line (a similar effect is incorporated in word-processing programs).

If you want to guess the whole word or phrase instead of keying in individual letters (you can do this at any point in the game), press 1. The program jumps to line 180. The word or phrase that you type in (T$) is compared to the stored string (A$). Then a "CORRECT" frame is PRINTed or if the guess is wrong, a "-WRONG-" frame is PRINTed. When a single letter is tried, lines 150 to 170 compare it to each character of the stored string in turn. If the guess is correct, the letter is PRINTed in the appropriate position in the display.

WORD GAME DISPLAY

```
HANGMAN - PRESS 1 TO MAKE A GUESS

M-C-OCOM-----

TRIES = 3

TRY A LETTER?
```

You can easily limit the number of guesses by adding the commands:

```
IF S>N THEN STOP
```

after the statements where the score S is calculated. If you make a mistake in keying in the program, it can be difficult to interrupt using the STOP and RESTORE keys, so to make this easier, add one extra line to check the value of T$:

```
145 IF T$="2" THEN STOP
```

To stop the program at any point, simply press 2.

WRITING GAMES 1

The next six pages will take you through writing a games program, showing you how to put all the phases together to build up a complete listing. Writing a games program requires some careful planning before you actually start writing lines. To begin with, you need to decide what sort of game you want. Many games combine your acquired skill with an element of chance (the roll of dice, the turn of a card and so on), and many have a number of different phases of play.

To plan a game, it's best to start by drawing a rough sketch of the screen display, marking the colors and positions of any fixed characters or patterns. You'll want to refer back to this as you write your program.

Next, you can draw up a flowchart showing the program steps and the order in which they will appear in the program. It isn't necessary to draw a detailed chart – a list of steps connected with arrows to show the order should be sufficient. A complete games program will be more complicated than anything you've written so far, so it is worth spending some time designing a program before you key it in.

Keying in the first phase of the game

With the game on this page, the planning stage has been completed, and you can now key in the first phase of what will be a two-stage program. The listing that follows is for a practical game – one that anyone should be able to play without any prior knowledge of the game or the computer. Below is the first screen of the program. This first phase of the game involves shooting at a moving spacecraft. As the program contains some user-defined characters and sprites, remember to move the BASIC program area out of the way by using the technique on page 14 to make room for them before you key the program in:

PHASE 1 SCREEN 1

```
10 PRINT CHR$(147) : POKE 53280,6
20 POKE 53281,12 : V=53248 : POKE V+21,3

30 POKE 2040,48 : POKE 2041,49
40 POKE V+39,7 : POKE V+40,2
50 FOR N=0 TO 127 : READ BYTE
60 POKE 3072+N,BYTE : NEXT N
70 POKE 56334,PEEK(56334) AND 254
80 POKE 1,PEEK(1) AND 251
90 FOR N=0 TO 511 : POKE 2048+N,PEEK(532
48+N) : NEXT N
100 POKE 1,PEEK(1) OR 4 : POKE 56334,PEE
K(56334) OR 1
110 POKE 53272,(PEEK(53272) AND 240)+2
120 FOR N=32 TO 39 : READ P
130 POKE 2560+N,P : NEXT N
140 S=53272 : POKE S+24,15
150 POKE S+5,0 : POKE S+6,240
160 POKE S,0 : POKE S+1,80
170 T$="000000" : N=0 : F=0 : Q=0
180 R=INT(RND(1)*15) : C=INT(RND(1)*26)
190 L=17 : M=16 : PRINT CHR$(147)

READY.
```

The program gives you a laser base which you can move left or right with the Z and X keys. You can fire, but only straight up the screen. A number of spacecraft approach you one by one, and you must destroy them to carry on. The program will start after you type RUN and then press any key. However, because it has a lot of DATA to POKE, it takes a little time to get started on screen. Don't assume that your listing is wrong if nothing appears for a few seconds.

PHASE 1 VARIABLES

The first phase of the game uses a total of 16 variables to control graphics and record hits on the target.

Variable	Function
A	Sets spacecraft direction
C$	Holds characters entered by player
F	Records total number of laser strikes
H	Flags successful laser strikes on target
L,M	Fix row and column co-ordinates of laser base
N,P	General variables
Q	Records the number of times laser fired
R,C	Fix row and column co-ordinates of spacecraft
S	SID chip base address
SC	Holds score for this phase
T	Sets delay loops
X,Y	Control co-ordinates of laser beam (1=right, −1=left)

The second screen of the program contains a number of lines which direct the computer to make decisions and then direct the computer to later subroutines. You will notice that as you go through the listing the line numbers sometimes jump by more than 10. This is because it's simpler to identify subroutines this way:

PHASE 1 SCREEN 2

```
200 N=N+1 : IF N=6 THEN 400
210 H=0 : A=4 : C=C+10
220 FOR I=1 TO RND(1)*900 : NEXT I
230 X0=INT((M+2)/32) : X1=INT((C+2)/32)
240 POKE V+16,X0+X1
250 POKE V,(M+2)*8-256*X0 : POKE V+1,(L+
7)*8
260 POKE V+2,(C+2)*8-256*X1 : POKE V+3,(
R+7)*8
270 POKE S+4,33 : FOR I=1 TO 20
280 NEXT I : POKE S+4,32
290 GET C$ : IF C$="Z" THEN M=M-1
300 IF C$="X" THEN M=M+1
310 IF M<2 THEN M=2
320 IF M>37 THEN M=37
330 IF C$="M" THEN GOSUB 500
340 IF M=1 THEN 180
350 X=C-C : C=C+A
360 IF C>35 OR C<4 THEN A=-A
370 F=INT(RND(0)*10+1)
380 IF P>9 THEN GOSUB 600
390 GOTO 230

READY.
```

Lines 290 and 300 let you move the laser base to either side with keys Z and X, and lines 310 and 320 stop it moving out of the screen. The laser moves one place for every key-press. If you key in POKE 650,128 as a direct command at the end of the listing, this will enable you to autorepeat the movement by holding the keys down.

If you press the M key, the program jumps to the "fire" routine at line 500. This PRINTs a line of user-defined characters. If the subroutine records a hit, the program begins a new attack.

Lines 350 and 360 control the movement of the spacecraft. Lines 370 and 380 make the program jump to the enemy attack routine at line 600 once in every 10 spacecraft moves. This is done by picking a random number from 1 to 10; only one of these numbers – 3 – will trigger the attack routine. Line 390 continues the same attack by jumping back to line 230.

The subroutine section

Finally, here is the last part of the program. It contains a pair of subroutines and the DATA for the graphics. Lines 510 to 560 draw and "undraw" the laser beam using the character defined in screen 1. If M is in the range from C−1 to C+1 then the laser has hit its target. Lines 600 to 700 control the return fire from the spacecraft. If it lands on the laser base, an explosion will sound and the Q value of your score will be altered.

Lines 400 and 410 calculate the score and end this phase of the program. The score is based on the time you take to complete the program, the number of times you fire the laser, and the number of direct hits. The score isn't actually used here, but it will appear later on as you develop the program.

Once you have typed in the listing on the following screens, SAVE it on a tape or disk before you RUN it so that if you have made any errors in keying, you will not lose the program if the machine "hangs", and you will be ready to combine it with the next phase.

PHASE 1 SCREEN 3

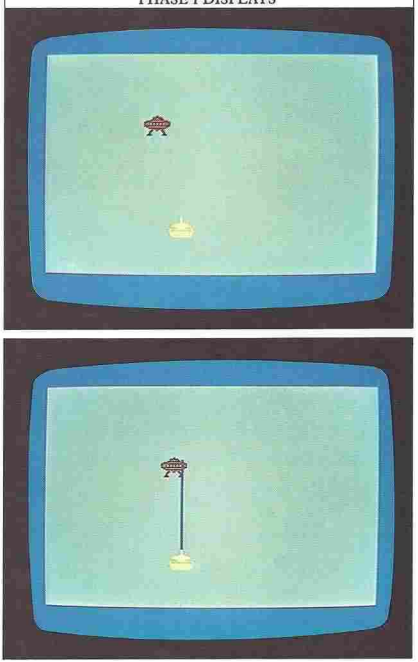
```
400 SC=TI/300+Q*2-10*F
410 STOP
500 Q=Q+1 : FOR Y=16 TO R STEP -1
510 POKE 214,Y : PRINT : POKE 211,M
520 PRINT CHR$(31);CHR$(101);
530 POKE S+4,33 : POKE S+4,32
540 NEXT Y : FOR Y=16 TO R STEP -1
550 POKE 214,Y : PRINT : POKE 211,M
560 PRINT CHR$(31);" ";: NEXT Y
570 IF M>C+1 OR M<C-1 THEN RETURN
580 POKE S+4,129 : FOR T=1 TO 200
590 NEXT T : H=1 : F=F+1 : POKE S+4,128
: RETURN
600 FOR Y=R+1 TO 17
610 POKE 214,Y : PRINT : POKE 211,X
620 PRINT CHR$(28);CHR$(101)
630 POKE S+4,33 : POKE S+4,32
640 NEXT Y : FOR Y=R+1 TO 17
650 POKE 214,Y : PRINT : POKE 211,X
660 PRINT CHR$(28);" ";: NEXT Y
670 IF X>M+1 OR X<M-1 THEN RETURN
680 POKE S+4,129 : FOR T=1 TO 200

READY.
```

PHASE 1 SCREEN 4

```
LIST 690-

690 NEXT T : Q=Q+10 : POKE S+4,128
700 RETURN
750 DATA 0,0,0,0,0,0,0,24
760 DATA 0,0,24,0,0,24,0,0
770 DATA 24,0,0,24,0,0,126,0
780 DATA 7,255,224,31,255,248,31,255
790 DATA 248,31,255,192,0,255,0,127
800 DATA 255,254,192,0,3,192,255,253
810 DATA 117,85,86,95,255,250,118,219
820 DATA 110,54,219,108,31,255,248,0
830 DATA 0,24,0,0,255,0,0,165,128
840 DATA 3,255,192,60,0,60,255,255,255
850 DATA 169,36,149,169,36,149,255,255
860 DATA 255,48,0,12,15,255,240,1,126
870 DATA 128,3,64,184,6,0,96,4,0,32
880 DATA 12,0,48,30,0,120,0,0,0,0
890 DATA 0,0,0,0,0,0,0,0,0,0
900 DATA 24,24,24,24,24,24,24,24
READY.
```

Here is the program in action. In the first display the laser is firing at the spacecraft, while in the second you can see the spacecraft retaliating:

PHASE 1 DISPLAYS



WRITING GAMES 2

In the second phase of the program, the scene changes from the air to the sea as a ship tries to depth-charge a moving submarine. Again, your aim is to hit the enemy to produce the best score. The scoring instructions are still not used in this phase, but are ready to be brought into operation when you key in the last phase of the game, linking up the first two parts.

You should type all three screens of listing for this phase on top of the program you SAVEd for phase 1. Remember once again to SAVE the combined program on a disk or tape before you RUN it. Because there are a large number of bytes POKEd into memory by the program, any typing mistakes that you make can POKE DATA into fatal places!

As before, the program uses a number of variables to control movement and subroutines.

PHASE 2 VARIABLES

The second phase uses a total of 6 variables to control movement and record hits.

Variable	Function
B,D	Submarine row and column co-ordinates (B=row, D=column)
C	Fixes column position of ship
F	Flag for "depth-charge dropped" (1=dropped, 0=not dropped)
U,E	Depth-charge row and column co-ordinates (U=row, E=column)

Setting the scene

The first section of the program produces a colored screen and sets up sprites for the ship and submarine:

PHASE 2 SCREEN 1

```
 165 S=54272 : GOTO 1000
 1000 FOR N=0 TO 127 : READ BYTE
 1010 POKE 3072+N,BYTE : NEXT N
 1020 V=53248 : PRINT CHR$(147);
 1030 POKE 2040,48 : POKE V+39,7
 1040 POKE 2041,49 : POKE V+40,0 : POKE V
 +21,3
 1050 FOR N=40 TO 55 : READ BYTE
 1060 POKE 2368+N,BYTE : NEXT N
 1065 POKE V+29,3
 1070 N=0 : F=0 : TI$="000000"
 1075 PRINT CHR$(154) : FOR P=1 TO 200
 1080 PRINT "■"; : NEXT P : PRINT
 1090 H=0*PEEK(V+31) : POKE V+39,7 : IF N
 =5 THEN 1500
 1100 POKE 53280,4 : POKE 53281,6
 1110 C=145 : B=100+INT(RND(0)*100)
 1120 D=INT(RND(0)*30)+50 : X0=0 : X1=0
 1130 GET C$ : IF C$="Z" THEN C=C-8
 1140 IF C$="X" THEN C=C+8
 1150 IF C>305 THEN C=305
 1160 IF C<30 THEN C=30

READY.
```

Don't worry for a moment about line 165 – you'll find out why it is included on the next page. Lines 1075 to 1080 PRINT a light blue sky over the dark blue sea. Line 1110 sets the column position for the ship and lines 1130 to 1190 control its movement across the water.

The aim of this game is to hit the submarine. The M key controls the release of the ship's depth-charges. The ship is also maneuverable, again being controlled by the Z and X keys. All these functions are controlled by a GET statement in line 1130 for a rapid response.

The ship always starts off in the middle of the screen. The position of your enemy is less predictable. Lines 1110 and 1120 set the random starting point for the submarine. It may appear at almost any position.

The main program and its subroutines

The second part of the listing contains most of the main program, together with a number of subroutines:

PHASE 2 SCREENS 2 AND 3

```
 1170 X1=INT(C/256) : POKE V+3,69
 1180 POKE V+C-X1*256
 1190 POKE V+16,X0+2*X1
 1200 IF F=1 THEN GOSUB 1310
 1205 IF W=1 THEN POKE V+39,0 : FOR T=1 T
 O 1000 : NEXT T : GOTO 1090
 1210 IF C$<>"M" OR F<>0 THEN 1230
 1220 F=1 : SC=SC+100 : GOSUB 1300
 1230 E=B-4 : X0=INT(D/256)
 1240 POKE V,D-X0*256 : POKE V+1,B
 1250 POKE V+16,X0+2*X1
 1260 IF D<10 THEN D=350
 1270 GOTO 1130
 1300 U=4 : E=(C-10)/8
 1310 POKE 214,U : PRINT : POKE 211,E
 1320 PRINT " ^ " : U=U+0.9
 1330 IF U>22 THEN F=0 : RETURN
 1340 POKE 214,U : PRINT : POKE 211,E
 1360 PRINT CHR$(144);CHR$(102)
 1370 IF (PEEK(V+31)AND1)=0 THEN RETURN
 1380 N=N+1 : F=0 : H=1
 1390 POKE S+4,129 : FOR T=1 TO 400

READY.
```

```
 1400 NEXT T : POKE S+4,128
 1410 POKE 214,U : PRINT : POKE 211,E
 1420 PRINT " ^ " : RETURN
 1500 SC=SC+TI
 1600 DATA 0,0,0,0,0,0,0,0,0,0,0,96,0
 1610 DATA 0,32,0,0,32,0,0,240,0
 1620 DATA 0,176,0,0,176,0,0,240,0
 1630 DATA 0,240,0,0,254,0,31,255,249
 1640 DATA 127,255,251,231,255,207
 1650 DATA 255,255,255,255,255,255
 1660 DATA 255,255,255,255,255,249
 1670 DATA 31,255,240,15,255,224,0
 1700 DATA 0,0,0,0,0,0,0,0,0,0,0,0
 1710 DATA 0,0,0,0,0,0,0,0,0,1,128
 1720 DATA 0,0,129,3,128,128,3,248,128
 1730 DATA 3,184,128,3,255,143
 1740 DATA 243,85,156,57,255,156
 1750 DATA 255,255,255,147,255,255
 1760 DATA 118,253,111,63,255,255
 1790 DATA 63,255,240,63,255,240,0
 1800 DATA 7,7,7,7,7,7,0,28,34
 1810 DATA 193,0,7,136,112,0
READY.
```

Line 1210 makes the program jump to the depth-charge routine at line 1300 if you have pressed M. The score, S, is also adjusted every time a depth-charge is dropped. The score is related to the time that has passed by using the jiffy clock in line 1500. Lines 1230 to 1260 control the movement and the appearance of the submarine. Line 1270 continues the program by returning it to line 1130 to check the keyboard for key-presses.

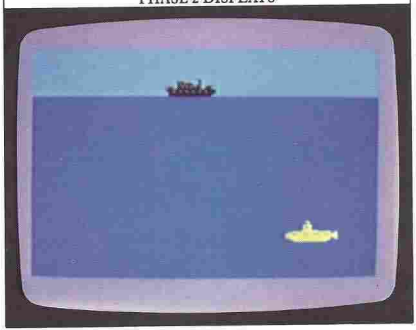
Lines 1300 to 1420 make the depth-charge travel down the screen. If the charge reaches the bottom of the screen, lines 1380 to 1420 reset F to its original value (zero) and then return to the main program. However, if the position of the depth-charge coincides with any position occupied by the submarine (sprite collision detection in line 1370) then the attack is terminated and a new one started.

You will notice that when you RUN the program only one depth-charge can be released at a time. If F has been set to 1 by line 1220, when a depth-charge is dropped, line 1210 stops you from dropping another one until F is once again equal to zero. This will be true either when the charge reaches the bottom of the screen (line 1330) or when it hits the submarine (line 1380).

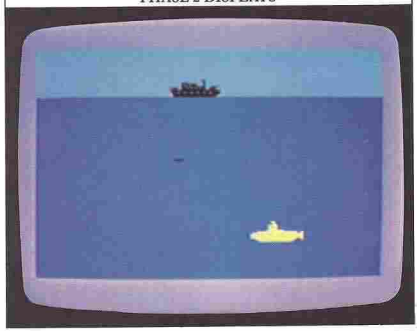
Lines 1600 to 1770 contain all the DATA necessary to program the sprites. The final two lines contain DATA which codes user-defined characters. The first character is the depth-charge. The second character is not used, but is available if you want to experiment with it. If you replace the reversed square character in line 1080 with CHR$(103) you will see a background of waves instead of solid sky. These waves are made up from single curved characters to give the impression of the water's surface. All the DATA in lines 1600 to 1810 is POKEd into memory by the loop of commands in lines 1000 to 1060.

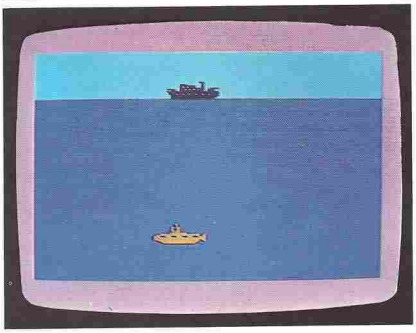
Here is a sequence of displays from the game; in the final screen the submarine has been hit, making it change color:

PHASE 2 DISPLAYS



PHASE 2 DISPLAYS





The continuing scoring routine

The time is once again used to calculate the score at the end of the game. If you want to check that the program is working properly, you can PRINT out your score for phase 2 with the following line:

```
1510 PRINT "PHASE 2 SCORE=";SC
```

Line 165 is included in this program to branch around the phase 1 program in memory. In the final version, you'll be taking out this line and adding some extra statements to combine the scores obtained in the two phases of the game. If you are really familiar with how the two phases work, you might like to look at the ranges of the score values they produce. Then see if you can think of a way to combine these so that they contribute to about half the overall score.

When you have made sure that the phase 2 program works you will be ready to link the two phases together and key in some playing instructions, making the games a single functional program.

WRITING GAMES 3

Now that you have keyed in and SAVEd the first two phases of the game, you are ready to add the game instructions and complete the part of the program which will produce your score. The extra line, 165, that you added to phase 2, now needs to be removed for the final version, so delete it now before moving on.

Adding the game instructions

If you RUN the program with the first two phases, you will find that although it's theoretically one program, it still behaves as two separate units. When you are writing games in phases like this, you will need to do a little tailoring to the final program to make it RUN through properly.

Linking the two phases is easily done. Change line 410 to:

```
410 POKE V+21,0:GOTO 2200
```

That's not a mistake, even though the sub-sinker program begins at line 1000. It's to allow you some space to add game instructions starting from line 2200. The new line 410 also turns off the sprites that are used in the first game.

Now you can go right back to the beginning and start the program off with a title frame containing all the instructions the player will need. The keys that control the objects moving on the screen need to be listed. You also need to tell the player how to start the game, bearing in mind that by the time the message appears, the program that contains the game will already be RUNning.

The next screen shows the instructions which appear before the first game. Line 5 makes the program jump over the two games to the instructions, and line 2140 makes the program go back to the beginning:

PHASE 1 INSTRUCTIONS

```
5 GOTO 2000
2000 PRINT CHR$(147) : POKE 53280,0 : PO
KE 53281,0 : PRINT CHR$(5)
2010 POKE 214,2 : PRINT : POKE 211,15
2020 PRINT CHR$(5);"POT SHOTS"
2030 PRINT TAB(15);"*********"
2040 POKE 214,8 : PRINT : POKE 211,9
2050 PRINT "FIVE ALIEN SPACECRAFT"
2060 PRINT TAB(5);"HAVE BEEN SIGHTED IN
 YOUR AREA" : PRINT
2070 PRINT TAB(5);"YOU MUST DESTROY THE
ALIENS" : PRINT
2080 PRINT TAB(5);"**********************
********"
2090 POKE 214,19 : PRINT : POKE 211,10
2100 PRINT "LASER BASE CONTROLS"
2110 PRINT TAB(8);"Z=LEFT  X=RIGHT  M=FI
RE" : PRINT
2120 PRINT TAB(9);"PRESS ANY KEY TO STAR
T"
2130 GET C$ : IF C$="" THEN 2130
2140 GOTO 10

READY.
```

Lines 2000 to 2120 PRINT the game title, and explain its controls. Instead of the program clearing the display after a time interval, it waits for a period that is controlled by a GET command. Line 2130 stops the computer from going any further by looping back on itself. This carries on until you enter a string which is not null – in other words, until you press any key. The condition for repeating line 2130 is then broken, and the program then goes to line 10, which clears the screen and starts the game:

PHASE 1 INSTRUCTIONS DISPLAY

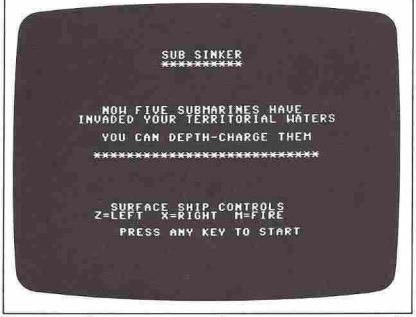
```
POT SHOTS
*********

FIVE ALIEN SPACECRAFT
HAVE BEEN SIGHTED IN YOUR AREA
YOU MUST DESTROY THE ALIENS
**************************

LASER BASE CONTROLS
Z=LEFT  X=RIGHT  M=FIRE
PRESS ANY KEY TO START
```

You can now key in the instructions for the second phase of the game. These work in the same way, and are activated at the end of the first phase by line 410. Again, GET is used to allow you to start the game only when you are ready. When you press any key, the program jumps to line 1000 which POKEs sprite DATA, and then the screen is cleared:

PHASE 2 INSTRUCTIONS

```
410 POKE V+21,0 : GOTO 2200
2200 PRINT CHR$(147) : POKE 53280,0 : PO
KE 53281,0 : PRINT CHR$(5)
2210 POKE 214,2 : PRINT : POKE 211,14
2220 PRINT CHR$(5);"SUB SINKER"
2230 PRINT TAB(14);"**********"
2240 POKE 214,8 : PRINT : POKE 211,7
2250 PRINT "NOW FIVE SUBMARINES HAVE"
2260 PRINT TAB(4);"INVADED YOUR TERRITOR
IAL WATERS" : PRINT
2270 PRINT TAB(7);"YOU CAN DEPTH-CHARGE
THEM" : PRINT
2280 PRINT TAB(6);"**********************
******"
2290 POKE 214,19 : PRINT : POKE 211,8
2300 PRINT "SURFACE SHIP CONTROLS"
2310 PRINT TAB(6);"Z=LEFT  X=RIGHT  M=FI
RE" : PRINT
2320 PRINT TAB(9);"PRESS ANY KEY TO STAR
T"
2330 GET C$ : IF C$="" THEN 2330
2340 GOTO 1000

READY.
```

PHASE 2 INSTRUCTIONS DISPLAY



You can of course use any keys you want to specify movement as long as you change them throughout the program. Now neither phase of the game will start until you are ready and press a key to begin.

Completing the scoring routine

Firstly, you need to add a routine to produce the score at the beginning of the program. The final score of the second phase is retained:

```
1500 LET SC=SC+TI
```

However, if you have played the two games independently and typed PRINT SC afterwards, you will have noticed that the first phase of the game yields a result ranging from −20 or so to several hundreds or several thousands. The results of the two games need to be roughly the same size. You can achieve this by multiplying the running score total in line 400 by 100.

```
400 LET SC=100*(TI/300+Q↑ 2−10*F)
```

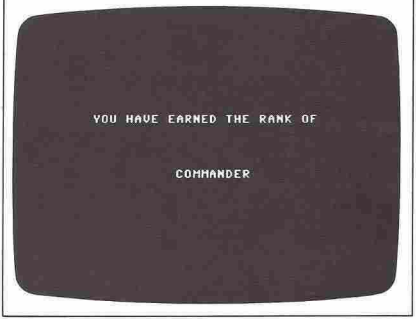
This line is a good test of your understanding of the variables from pages 46–49! To make the presentation of the score more interesting, you can add a few lines to turn this purely numerical score into a ranking. This is quite a useful technique in games programs, because it gives a new player some idea of how the program rates his or her skill. It's much better than a purely numerical result which gives you no idea how your score compares with the complete range that the program is likely to produce.

This ranking feature is very often used in a whole range of programs. Even though adding these little extra touches is fairly simple, it is these small additions to the main sequence that can make the difference between an average program and one that you can be really proud of. Here's the scoring section and one of the displays it produces after the complete game has been played:

SCORING ROUTINE

```
LIST
1510 POKE V+21,0 : GOTO 2400
2400 SC=SC/5 : PRINT CHR$(147)
2410 POKE 53280,0 : POKE 53281,0 : PRINT
 CHR$(5)
2420 POKE 214,6 : PRINT
2430 PRINT TAB(6);"YOU HAVE EARNED THE R
ANK OF"
2440 IF SC<1000 THEN A$="COMMANDER"
2450 IF SC>=1000 THEN A$="CAPTAIN"
2460 IF SC>=2000 THEN A$="PILOT"
2470 IF SC>=4000 THEN A$="CADET"
2480 IF SC>=6000 THEN A$="ROOKIE"
2490 POKE 214,12 : PRINT : POKE 211,16
2500 PRINT A$
2510 GOTO 2510
READY.
```

SCORE DISPLAY



Lines 2440 to 2480 divide the scores up into bands, each of which is assigned to a rank. A series of IF ... THEN lines decides where your score comes in the ranking. You can change the cut-off scores for each band to make the games easier or harder (if you're feeling ambitious you can actually program this as a difficulty option).

You have now completed a two-phase game with instructions, action and a scoring routine. Although the two phases used on these pages are relatively simple, the way that they are combined can be used to build up games of your own that are much more complex. You can use this multi-phase technique to put together a number of sub-programs, each written and tested independently. The only restriction on this is the size of the computer's memory, but unless you are combining long programs, you shouldn't have any problems.

All you need now is practice. The best way to get this, as a beginner, is to take an existing program such as the one you have seen over the last six pages, and then to customize it in your own way.

FILING DATA WITH ARRAYS

An array is a way of storing a collection of facts and/or figures in the computer's memory in the form of a table, so that you can locate any item in the table without having to go through all the others first. Each item in an array is specified by one or more numbers. In the following array, each item is given a pair of co-ordinates which identify it and nothing else:

	0	1	2	3	4	5
0	FRED	KATE	JOHN	JANE	ALAN	JUDY
1	100	250	840	125	223	691

This is a 6×2 array, so-called because it has 6 columns by 2 rows. Item (1,1) is 250, item (2,0) is JOHN, and so on. Because two numbers are needed to identify each item, this array is known as a two-dimensional array. If it was composed of only one row of names or numbers, it would need only one number to identify each item and so it would be a one-dimensional array. The BASIC keyword DIM is used to tell the computer how big an array is to be, by specifying the largest subscript (the highest position) in each dimension of the array.

What use are arrays?

A one-dimensional array can store a list of frequently used numbers or strings:

ONE-DIMENSIONAL ARRAY PROGRAM

```
LIST
 10 DATA "JANUARY","FEBRUARY","MARCH"
 20 DATA "APRIL","MAY","JUNE","JULY"
 30 DATA "AUGUST","SEPTEMBER","OCTOBER"
 40 DATA "NOVEMBER","DECEMBER"
 50 DIM M$(11)
 60 FOR N=0 TO 11
 70 READ M$(N)
 80 NEXT N
 90 PRINT CHR$(147)
100 FOR N=0 TO 11
110 POKE 214,N+4 : PRINT : POKE 211,14
120 PRINT M$(N)
130 NEXT N
READY.
```

Here line 50 tells the computer that the array M$ has 12 entries (notice that array subscript numbers start at 0 so the array elements are numbered 0 to 11). This program PRINTs out a list of the months of the year given in lines 10 to 40. Although there are easier ways of doing this, later on in a program you may want to match up a month with other information or the result of calculations. Using this listing, you can pick out any month by using M$(N) where N is the month number.

When the program is RUN, the display it should produce looks like this – a month chart ready for more information:

ONE-DIMENSIONAL ARRAY DISPLAY

```
JANUARY
FEBRUARY
MARCH
APRIL
MAY
JUNE
JULY
AUGUST
SEPTEMBER
OCTOBER
NOVEMBER
DECEMBER

READY.
```

Writing tables with arrays

Now you can build upon this calendar array program to make it do something useful. Add a second array, a numeric array, so that you can list some totals or values against each month:

TWO-DIMENSIONAL ARRAY PROGRAM

```
LIST
 10 DATA "JANUARY","FEBRUARY","MARCH"
 20 DATA "APRIL","MAY","JUNE","JULY"
 30 DATA "AUGUST","SEPTEMBER","OCTOBER"
 40 DATA "NOVEMBER","DECEMBER"
 50 DATA 2.5,1.0,2.3,7.5,9.5,4.5
 60 DATA 3.5,4.0,4.5,4.5,4.0,2.5
 70 DIM M$(11),R(11)
 80 FOR N=0 TO 11
 90 READ M$(N) : NEXT N
100 FOR N=0 TO 11
110 READ R(N) : NEXT N
120 PRINT CHR$(147)
130 POKE 214,3 : PRINT : POKE 211,10
140 PRINT "MONTH          RAINFALL"
150 FOR N=0 TO 11
160 POKE 214,N+5 : PRINT
170 PRINT TAB(9);M$(N);TAB(22);R(N)
180 NEXT N
READY.
```

The table now has two headings. You don't have to PRINT all the members of the string array – M$(N) – before moving on to select the numeric array – R(N). Line 170 takes one item from each array. As these are to be PRINTed on consecutive rows of the screen, they can be easily identified by relating them to the row number. For each value of N, M$(N) and R(N) are PRINTed at different TAB positions along row (N+5):

TWO-DIMENSIONAL ARRAY DISPLAY

```
          MONTH          RAINFALL

        JANUARY            2.5
        FEBRUARY           1
        MARCH              5.25
        APRIL              7.5
        MAY                9.25
        JUNE               4.5
        JULY               3.5
        AUGUST             4
        SEPTEMBER          4.25
        OCTOBER            4.5
        NOVEMBER           4
        DECEMBER           2.5

READY.
```

Adding an extra dimension

Once you have understood the rainfall program, you can be more ambitious by constructing a much more complicated table. In the financial planning program below, the columns in the display are interrelated and you have the option of changing some of the information displayed by keying in a new tax rate.

Lines 10 and 20 contain the DATA for the first part of the array, a series of prices, and line 30 the DATA for a second part – a series of quantities. Line 40 contains some co-ordinates which will be used later in the program. Lines 50 to 80 dimension the 9×2 array and READ in its DATA. Lines 210 to 370 simply DRAW the grid of lines that frames the DATA. The co-ordinates of the bottom ends of the vertical lines are stored in line 40 and are used in a 7×1 array.

The DATA is PRINTed in the grid by lines 120 to 200. It is PRINTed every line from rows 6 to 14 (this is set by line 130):

TAX TABLE PROGRAM

```
LIST -180

10 DATA 1.98,2.40,5.60,1.05,4.35
20 DATA 2.99,1.92,7.20,5.45,2
30 DATA 20,19,11,45,15,15,24,4,6
40 DATA 3,8,13,16,23,28,34
50 T=15  : DIM Q(8,1),X(6)
60 FOR C=0 TO 1  : FOR R=0 TO 8
70 READ Q(R,C)  : NEXT R  : NEXT C
80 FOR C=0 TO 6  : READ X(C)  : NEXT C
90 PRINT CHR$(147)
100 POKE 214,4  : PRINT  : POKE 211,4
110 PRINT "ITEM  COST  NO  SUB/TL  TAX  TOTA
L"
120 FOR N=0 TO 8
130 POKE 214,N+6  : PRINT
140 POKE 211,4  : PRINT N+1;
150 POKE 211,8  : PRINT Q(N,0);
160 POKE 211,13  : PRINT Q(N,1);
170 POKE 211,18  : PRINT Q(N,0)*Q(N,1);
180 POKE 211,23  : PRINT (INT((Q(N,0)*Q(
N,1)*T/100)+0.005)*100))/100;

READY.
```

TAX TABLE PROGRAM

```
190 POKE 211,28  : PRINT (INT(((Q(N,0)*Q(
N,1)*(1+T/100))+0.005)*100))/100;
200 NEXT N
210 FOR C=0 TO 6  : FOR Y=4 TO 14
220 POKE 214,Y  : PRINT  : POKE 211,X(C)
230 PRINT CHR$(98)  : NEXT Y  : NEXT C
240 FOR C=4 TO 13
250 POKE 214,3  : PRINT  : POKE 211,C
260 PRINT CHR$(99)
270 POKE 214,15  : PRINT  : POKE 211,C
280 PRINT CHR$(99)
290 POKE 214,5  : PRINT  : POKE 211,C
300 PRINT CHR$(99)
310 NEXT C  : FOR C=0 TO 6
320 POKE 214,3  : PRINT  : POKE 211,X(C)
330 PRINT CHR$(178)
340 POKE 214,5  : PRINT  : POKE 211,X(C)
350 PRINT CHR$(123)
360 POKE 214,15  : PRINT  : POKE 211,X(C)
370 PRINT CHR$(177)  : NEXT C  : PRINT
380 PRINT TAB(8);"TRY A NEW TAX RATE";
390 INPUT T  : GOTO 90
READY.
```

The last two items PRINTed by lines 180 and 190 look particularly complex. If the subtotal was 8.25, the tax would be calculated as 0.15*8.25=1.2375 – too many decimal places. To solve that, the tax is multiplied by 100, the INTeger value of it is taken (removing all the decimal places) and it is divided by 100 again. The 0.005 is added to ensure that the final figure is rounded down to the nearest unit to fit into the table.

Lines 380 and 390 invite you to enter a new tax rate. If you do and press RETURN, all the figures in the table that use the tax rate are recalculated. This instant recalculation facility is the principle behind a type of financial planning program called a spreadsheet. Interrelated columns of figures representing income, raw material/production costs, overheads and so on can be entered. Then the effects of changing one or more of these parameters can be observed as all the totals are recalculated throughout the display (you can also modify this sort of program so that the initial information for the table can be INPUT):

TAX TABLE DISPLAY

ITEM	COST	NO	SUB/TL	TAX	TOTAL
1	1.98	20	39.6	5.94	45.54
2	2.4	19	45.6	6.84	52.44
3	5.6	11	61.6	9.24	70.84
4	1.05	45	47.25	7.09	54.34
5	4.35	15	65.25	9.79	75.04
6	2.99	15	44.85	6.73	51.58
7	1.92	24	46.08	6.91	52.99
8	7.2	4	28.8	4.32	33.12
9	5.45	6	32.7	4.91	37.6

TRY A NEW TAX RATE? 12.5

TRACING ERRORS

Even when you plan a program meticulously and take every care when keying it in, you may still find that it refuses to RUN properly. On these two pages you can take a look at how to debug a program. You've seen the program before, it's the word-game program from page 45, but this time it has eight serious bugs in it. It's as if the program has been written and keyed in hurriedly so that it will not work. Don't cheat by looking back at the earlier program! See if by checking the listing any of the bugs become obvious to you then see if what you think is wrong is corrected on these two pages.

Trying a test RUN

When you RUN a bugged program, you'll discover its bugs in two ways. Firstly, any lines that don't make sense to the computer will produce error messages, and secondly problems in structure or detail will show up in the way it RUNs:

WORD GAME PROGRAM

```
10 POKE 53280,0 : POKE 53281,0 : PRINT C
HR$(147)
20 POKE 214,8 : PRINT : PRINT TAB(5);"AS
K A FRIEND TO TYPE A WORD"
30 POKE 214,10 : PRINT : PRINT TAB(5);"O
R PHRASE, FOR YOU TO GUESS"
40 POKE 214,15 : PRINT : PRINT "   ** DO
N'T LOOK AT THE SCREEN! **"
50 PRINT : PRINT "  PRESS RETURN WHEN YO
U'VE FINISHED"
60 PRINT TAB(10); : INPUT "WORD = ";A$
70 PRINT TAB(2); "HANGMAN - PRESS 1 TO M
AKE A GUESS"
80 P=(32-LEN(A$))/2 : S=0 : POKE 214,7 :
 PRINT
90 FOR N=1 TO LEN(A$) : POKE 211,P + ";
100 IF MID$(A$,N,1)=" " THEN PRINT " ";
: NEXT N
110 PRINT "-"; : NEXT M
120 POKE 214,18 : PRINT : PRINT TAB(9);
: INPUT "TRY A LETTER";T$
READY.
```

```
LIST 130-

130 IF T$="I" THEN 180
140 S=S+1 : POKE 214,13 : PRINT : PRINT
TAB(2);"TRIES =";S : POKE 214,7
150 PRINT : FOR N=1 TO LEN(A$)
160 IF T$=MID$(A$,N,1) THEN POKE 211,P-N
: PRINT T$
170 NEXT N : GOTO 120
180 POKE 214,18 : PRINT : PRINT TAB(7);
: INPUT "ENTER YOUR GUESS";T$
190 PRINT CHR$(147)
200 POKE 214,10 : PRINT : POKE 211,14
210 IF T$=A$ THEN PRINT "CORRECT"
220 IF T$<>A$ THEN PRINT "WRONG-"
230 FOR T=1 TO 5 : NEXT T : GOTO 10
READY.
```

When you try to RUN the program, you'll find that the title frame comes up and then you are asked to enter a word. Throughout this debugging session, try keying in "TRACING ERRORS" – this will enable you to get the same results as the ones shown here.

When you key these two words in, you'll find that you get an error message straightaway:

```
?NEXT WITHOUT FOR ERROR IN 110
```

This means that the program stopped when it encountered something that it didn't understand in line 110. If you LIST 70–140, you will see that a NEXT statement containing the variable M appears in line 110. Looking back through the program for a matching FOR statement reveals that the FOR in line 90 uses the variable N and not M. Between these statements, you can see that the variable N appears, suggesting that this is the correct one and that M is a mistake in keying in. To make the loop work properly, change NEXT M to NEXT N. Now try the program again:

CRASHED PROGRAM DISPLAY

```
ASK A FRIEND TO TYPE A WORD
OR PHRASE, FOR YOU TO GUESS

** DON'T LOOK AT THE SCREEN! **
PRESS RETURN WHEN YOU'VE FINISHED
TRY A LETTER?C NG ERRORS
HANGMAN - PRESS 1 TO MAKE A GUESS
```

This time, the title frame and the test string entry work properly, but the title frame stays on the screen when the next phase of the game starts. That's easily dealt with by adding:

```
PRINT CHR$(147)
```

to the start of line 70.

Another problem that you can see with the display above is that the characters PRINTed to represent the two words all seem to have appeared in the same position.

Looking carefully around the loop that PRINTs these characters, you might notice that their X position is determined by POKE in line 90. You can see that the value POKEd is the constant value of the variable P,

whereas you would expect the value to vary with the value of N. You can make it do this by changing the value POKEd in line 90 from P to P+N.

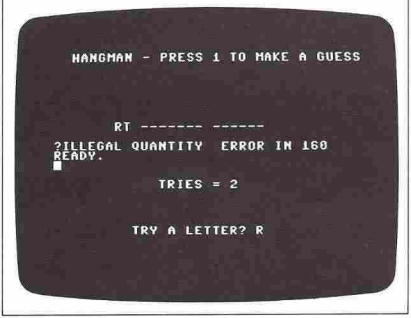
How to track down more bugs

Now when you RUN the program, as soon as you key in your first guess at a letter, you will get another error message:

```
?TYPE MISMATCH ERROR IN 150
```

This means that a number or numeric expression was found in a position in a line where a string value or expression was expected, or vice versa. LIST the line in the error message, and you will find that it uses the LEN function. The job of this function is to return the length of the string contained within its brackets. If you now look at the contents of the brackets you will see the variable A. This is numeric and not string, and so you have found the cause of another problem. To cure it, replace A by A$, and try the program again:

CRASHED PROGRAM DISPLAY 2



This time, the program allows you to enter the two guesses T and R before it crashes again, this time with the message:

```
?ILLEGAL QUANTITY ERROR IN 160
```

This message is given when a parameter has gone out of the range of a command. Add to this the fact that the display shows the guesses T and R PRINTed in the wrong positions and in reverse order, and you will have good reason to suspect that the POKE in line 160 is the cause of the problem. To check this, get the computer to PRINT the value being POKEd:

```
PRINT P-N
```

You should find that this gives a negative result, which is certainly illegal as a value to be POKEd. As the letters are being PRINTed in the wrong direction, the value of N should be added to P and not subtracted:

CRASHED PROGRAM DISPLAY 3

You should now see that when you enter T and R as guesses, the letters are entered on the screen in the correct horizontal positions and are PRINTed in the right order. Unfortunately, you will have uncovered another bug – multiple occurrences of a letter are not PRINTed on the same line. It looks as though a RETURN is being put in after each character.

This bug is easy to track down to the PRINT statement at the end of line 160, which should terminate with a semi-colon to suppress RETURN.

RUNning the program again after correcting this produces results that are fine as you key in the letters T, R, A and C, but when you key in the letter I, the program stops asking for letter guesses and asks you to try for a whole word guess. According to the instructions, this should only happen if you key in the number 1 as a letter guess. But the number 1 and the letter I look very similar, and as you'll see if you check line 130, the programmer has got them the wrong way around, a fault which is easily corrected.

The final bug is fairly straightforward. When you enter some letter guesses and then try to guess the whole thing, the program should tell you whether the guess is right or wrong. But the last bug doesn't allow this message to stay on the screen long enough for you to read it. To cure this last problem, increase the time delay loop in line 230 from 5 to 5000. Now you can check with the program on page 45 to confirm that the two programs now RUN in the same way.

Ways to avoid writing bugged programs

As you develop your own programs, constant checking should prevent all but a few bugs from slipping into the final listing. When you're testing a program that you've written, put it through all the situations that it will meet in use, particularly testing any numerical limits. If it's supposed to have safeguards to stop it crashing in some circumstances, test them too.

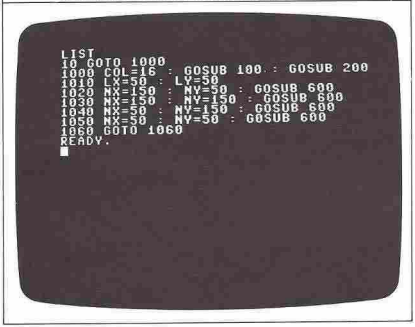
HINTS AND TIPS

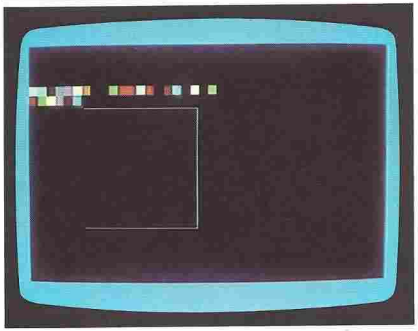
One of the biggest problems that you have probably come across during this book is typing listings. It's very difficult to get a listing fully correct the first time. Normally this shows up in an error message, but when you are using high resolution, error messages are unreadable. Your program stops, and all you can see of the error message is a row of colored squares. What can you do to find out what is wrong?

Identifying bugs in high resolution

The next screen shows a graphics program which uses the high-resolution subroutines, and after that is the display. As you can see, there is a problem somewhere:

BUGGED HIGH-RESOLUTION PROGRAM



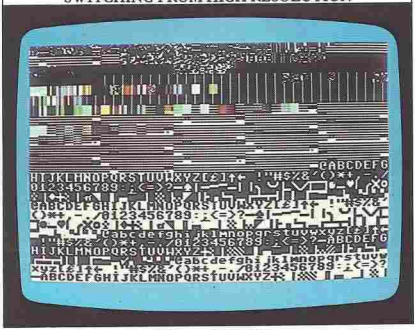


Because the area of memory that usually holds text now holds the color memory, you can't get letters on the high-resolution screen. The normal way to go back to the text resolution display is with RUN STOP and RESTORE. However, this results in the screen being cleared so the error message disappears as well, leaving you no wiser as to why your program crashed. What you need is a method of switching from high resolution back to text resolution without clearing the screen. You can do this by entering the following commands directly after the error message has appeared. First, key in this line:

```
POKE 53272,PEEK(53272) AND 247
```

The screen should immediately fill up with a display like this:

SWITCHING FROM HIGH RESOLUTION

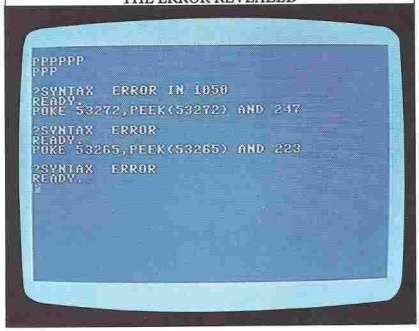


Now key in a second line of POKEs to bring the error message onto the screen:

```
POKE 53265,PEEK(53265) AND 223
```

This screen shows what you should see next:

THE ERROR REVEALED



As you can see, this does work. However, there is one problem. The two sets of POKEs are rather complex and you have to type them into the machine "blind", as you cannot see what you are typing on the screen.

A better solution is to accept the fact that you will probably make mistakes while developing programs, but prepare for them in advance. What you do is to key in the previous two lines, plus an END statement, into your high-resolution graphics subroutines. It's done like this:

```
20 POKE 53272,PEEK(53272) AND 247
30 POKE 53265,PEEK(53265) AND 223
40 END
```

Then if your high-resolution graphics program crashes, all you need to do is type in GOTO 20 and the error message will appear.

How to make RESTORE more useful

One facility which is absent from the Commodore's BASIC repertoire is the ability to RESTORE the machine's DATA pointer to any given DATA statement in a program. For example, you can't type:

```
10 RESTORE 50
```

meaning "reset the DATA pointer to the beginning of line 50 rather than the first line of DATA in the program". This facility can be very useful where a lot of text messages are to be stored, such as in an accounts program or an adventure game, and need to be accessed quickly. The big advantage of this is that no memory space is needed to hold the DATA other than in the DATA statements in the program. You don't need to dimension an array to READ the DATA into, because it can be READ straight from the DATA statements.

The following screen gives a subroutine listing, starting at line 5000, which will POKE a short machine code routine into a free area of the computer's memory:

MACHINE CODE RESTORE ROUTINE

```
LIST

5000 FOR C=0 TO 23
5010 READ A
5020 POKE 49494+C,A
5030 NEXT C
5040 DATA 165,62,133,20,165,64
5050 DATA 133,21,32,19,166,165
5060 DATA 99,233,1,133,65,165
5070 DATA 96,233,0,133,66,96
READY.
```

This block only needs to be carried out once in a program. It contains instructions to get the computer to change the DATA pointer to the next DATA statement in use.

The next screen shows a short subroutine which uses this block of machine code to produce the effect of RESTOREing to a line number:

USING THE RESTORE ROUTINE

```
LIST

5500 LH=INT(RN/256)
5510 LL=RN-LH*256
5520 POKE 63,LL
5530 POKE 64,LH
5540 SYS 49494
5550 RETURN
READY.
```

The entry point for this subroutine is at line 5500. You can set the variable RN to the line number you wish to RESTORE to and then make the computer GOSUB 5500.

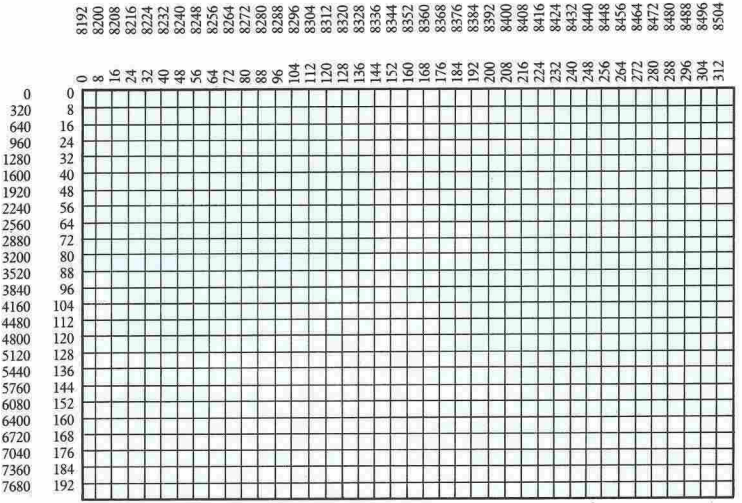
Where to store machine code subroutines

As you have just seen, it is often helpful to access a short piece of machine code from within a BASIC program. Usually, this is done with the BASIC keyword SYS. This is followed by a number which is the address in memory of the start of the machine code routine. The SYS keyword is very like the BASIC statement GOSUB, in that after the machine code routine, the program goes back to the statement following SYS. To make sure that the program returns to the correct point, the machine code routine must end with the machine code equivalent of RETURN, which is RTS (ReTurn from Subroutine). This instruction has a decimal value of 96, which can be seen at the end of a machine code DATA list.

When you want to use machine code subroutines as part of your BASIC programs, one of the problems which you may encounter is deciding where in memory to locate the bytes that make up the machine code. On the Commodore this problem is easy to solve. There is a RAM area from addresses 49152 to 53247, that is 4K in all, which is available for machine code and which is unused by anything else within the computer. This area is ideal for storing machine code subroutines. The RESTORE subroutine on the left is located in this "safe" area of memory.

HIGH-RESOLUTION GRIDS

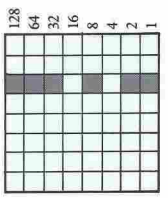
The two grids here enable you to identify the high-resolution memory location for any pixel or group of pixels on the screen. The first grid has two sets of numbers along each side. The innermost numbers are simple horizontal and vertical co-ordinates. The outermost numbers allow you to work out memory locations for each square on the grid. The pixels in each square are controlled by eight consecutive locations in the memory. To find the number of the lowest location in the sequence of eight, add together the horizontal and vertical location numbers on the grid. You can then move on to the 8 × 8 grid below it.



How to set individual pixels

Once you have established which eight memory locations control the square you have picked out, you can then POKE values into them to light individual pixels. Each memory location controls just one row of pixels, so working downward from the top of the square, there are eight separate locations involved. In the grid on the right, just six pixels in line 2 are being set. To do this, you would have to work out the location that starts the square, using the grid above, and then add 2 to it. This gives you the right location for line 2. Then you need to add up the pixel values – 128+64+32+8+2+1 in this example – and POKE them into this memory location. If you wanted to light pixels in more than one row of this square, you would need to POKE more than one location.

Working out high-resolution memory locations like this is a useful way of getting to know exactly how the Commodore operates in high resolution. If you try it out, it will give you an idea of how the high-resolution sub-routines featured earlier in this book actually function.



SPRITE GRIDS

Commodore sprites are each composed of 21 rows of 24 pixels, each pixel being controlled by a single bit in a POKE statement. You can use the grid below to design your own sprites and work out their POKE values. These can then be entered as DATA in a program. After pencilling in a design on the grid, add up the bit values for each group of eight pixels on a row. You can then record the totals in the columns on the right.

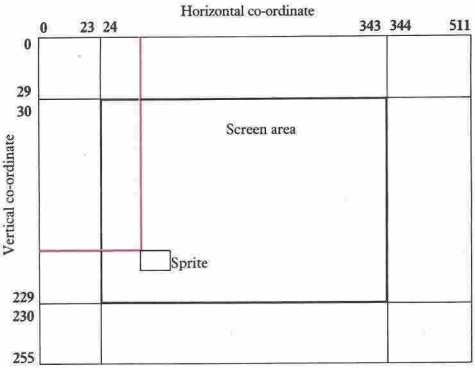
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

DATA values		

Positioning sprites

Sprite position co-ordinates can have any value from 0–255 vertically or 0–511 horizontally. However, only a part of this range is actually visible on the screen. The diagram on the right shows how these co-ordinates relate to the screen. The central panel is the visible area on the screen. This extends from 24 to 343 horizontally and 30 to 229 vertically. This means that you can move sprites on and off the screen smoothly.

Because vertical position runs from 0 to 255, it can be controlled by a single byte of information. Horizontal position, on the other hand, needs two bytes if the whole of the range from 0 to 511 (=2×255) is to be used. If you want a sprite to continue moving past horizontal position 255, the V+16 location must be turned on for the sprite concerned.



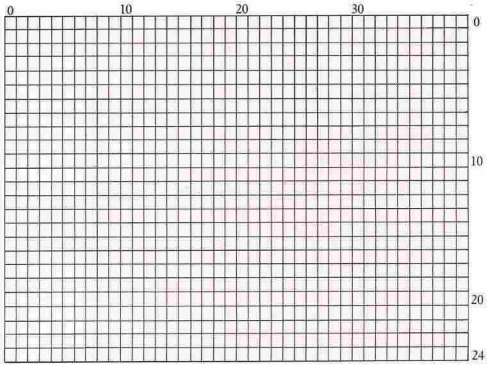
SCREEN MEMORY CODES

At text resolution, every point on the Commodore screen can be accessed using the POKE command. This allows you to put characters in any color onto the screen at specified positions. Two separate POKE commands are needed – one to specify the character that is to appear at a screen position, and another to set its color. The grid below shows how the screen memory and color memory locations are selected.

Screen memory and color memory

To find out the two memory locations of any point on the screen, first find out the row POKE values. The complete screen and color memory locations can then be determined by adding the column number to each POKE value. Once you have determined these two numbers, you can add a character and color value.

1024	55296
1064	55336
1104	55376
1144	55416
1184	55456
1224	55496
1264	55536
1304	55576
1344	55616
1384	55656
1424	55696
1464	55736
1504	55776
1544	55816
1584	55856
1624	55896
1664	55936
1704	55976
1744	56016
1784	56056
1824	56096
1864	56136
1904	56176
1944	56216
1984	56256



Character POKE values

The chart below shows the POKE values for the Commodore's upper case and lower case characters. Character codes are POKEd together with the screen memory values. Values from 128 to 255 produce the reverse images of the characters shown here.

	0	1	2	3	4	5	6	7	8	9
0	@	A a	B b	C c	D d	E e	F f	G g	H h	I i
10	J j	K k	L l	M m	N n	O o	P p	Q q	R r	S s
20	T t	U u	V v	W w	X x	Y y	Z z	[	£	]
30	↑	←	SPACE	!	"	#	$	%	&	'
40	(	)	*	+	,	−	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?		♠ A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	♥ S	T	U	V	W	♣ X	Y
90	♦ Z				π		SPACE			
100										
110										
120			↙							

ASCII CHARACTER SET

The ASCII character set forms a single sequence of characters and control functions that can be accessed by the command CHR$. The ASCII system provides a standard digital coding for computer characters. The codes from 33 to 127 represent the same characters on almost all microcomputers, while the codes outside this range are used differently by different machines. On the Commodore these control a range of functions like color settings and represent keyboard graphics characters. The ASCII code for each character only uses 7 bits of a byte, leaving room for an eighth bit for "parity checking", or transmission error monitoring.

	0	1	2	3	4	5	6	7	8	9
0						White			Disables SHIFT+C=	Enables SHIFT+C=
10				RETURN	Lower case			Cursor down	RVS ON	CLR HOME
20	INST/DEL								Red	Cursor right
30	Green	Blue	SPACE	!	"	#	$	%	&	.
40	(	)	*	+	,	−	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	£	]	↑	←				
100										
110										
120										Orange
130				Function key 1	Function key 2	Function key 3	Function key 4	Function key 5	Function key 6	Function key 7
140	Function key 8	SHIFT + RETURN	Upper case		Black	Cursor up	RVS OFF	CLR HOME	INST/DEL	Brown
150	Light red	Light gray	Medium gray	Light green	Light blue	Dark gray	Purple	Cursor left	Yellow	Cyan
160	SPACE									
170										
180										
190										
200										
210										
220					SPACE					
230										
240										
250										

GLOSSARY

Entries in **bold type** are BASIC keywords.

ABS
Gives the absolute value of a number.

AND
Combines two conditions or numbers, giving a result of 1 only if both conditions or numbers have a value of 1.

ASC
Gives the ASCII code of a character.

ASCII
American Standard Code for Information Interchange; the character coding system used by the Commodore.

BASIC
Beginners' All-purpose Symbolic Instruction Code; the most commonly used high-level programming language.

Binary
A counting system used by computers based on only two numbers – 0 and 1.

Bit
A binary digit – 0 or 1.

Byte
A group of eight bits.

Chip
A single package containing a complete electronic circuit. Also called an integrated circuit (IC).

CHR$
Converts an ASCII code into the character it represents.

COS
Gives the cosine of an angle.

CPU
Central Processing Unit. Normally contained in a single chip called a microprocessor, this carries out the computer's arithmetic and controls operations in the rest of the computer.

Cursor
A flashing symbol on the screen, showing where the next character will appear.

DATA
The computer treats whatever follows **DATA** as information that may be needed later in the program. Used in conjunction with **READ**.

Debugging
The process of ridding a program of errors or bugs.

DEF FN
Defines a function.

DIM
Informs the computer about the size of an array.

END
Halts a program. (See also **STOP**.)

Envelope
The change in amplitude (volume) of a note while it is playing. Envelope shapes are selected with **POKE**.

Filename
A name given to a program or set of data to enable storage and recall on a tape or disk.

FN
Indicates that the variable following represents a function. (See also **DEF FN**.)

FOR ... NEXT
A loop which repeats a sequence of program statements a specified number of times.

GOSUB
Makes the program jump to a subroutine beginning at the line number following the command. The subroutine must always be terminated by **RETURN**.

GOTO
Makes a program jump to the line number following the command.

IF...THEN
Prompts the computer to take a particular course of action only if the condition specified is detected.

INPUT
Instructs the computer to wait for some data from the keyboard which is then used in a program.

INT
Converts a number with a decimal fraction into a whole number by rounding down.

K
Abbreviation of kilobyte (1024 bytes).

LEFT$
Forms a string from the left-hand part of another string.

LET
Assigns a value to a variable. The use of **LET** is optional on the Commodore.

LEN
Counts the number of characters in a string.

LIST
Makes the computer display the program currently in its memory.

LOAD
Transfers a program from a tape or disk into the computer's memory. The program is identified by a filename.

Loop
A sequence of program statements which is executed repeatedly or until a specified condition is satisfied.

MID$
Forms a string from the middle part of another string.

NEW
Removes a program from the computer's memory.

ON ... GOTO/GOSUB
Makes a program jump to one of a number of statements or subroutines depending on the value of a variable.

OR
Combines two conditions or numbers, giving a result of 1 if either of the conditions or numbers has a value of 1.

PEEK
Reads the numeric value in a specified memory location.

POKE
Puts a numeric value into a specified memory location.

PRINT
Makes whatever follows appear on the screen.

READ
Instructs the computer to take information from a **DATA** statement.

REM
Enables the programmer to add remarks to a program. The computer ignores whatever follows **REM** in a program statement.

RESTORE
Resets the computer to **READ** the first item in a **DATA** list.

RETURN
Terminates a subroutine. (See also **GOSUB**.)

RIGHT$
Forms a string from the right-hand part of another string.

RND
Produces numbers at random within specified limits.

SAVE
Records a program currently in memory onto a tape or disk. The program is identified by a filename.

SGN
Tests the sign of a number.

SID
Sound Interface Device; the chip used by the Commodore to produce sound.

SIN
Gives the sine of an angle.

Sprite
A mobile object block that is defined using **POKE**.

SQR
Produces the square root of the number that follows it.

STEP
Sets the step size in a **FOR ... NEXT** loop.

STOP
Halts a program and **PRINT**s out the line number in which it appears.

String
A sequence of characters treated as a single item – someone's name, for instance.

Subroutine
A part of a program that can be called when necessary, to produce a particular display or carry out a number of calculations repeatedly, for example.

SYS
Gives the starting location of a machine code program.

TAB
Positions text along a line.

Variable
A labeled slot in the computer's memory in which information can be stored and retrieved later in a program.

VERIFY
Checks that a program has been recorded correctly on a tape or disk using **SAVE**.

VIC
Video Interface Circuit; the chip responsible for controlling sprites.

INDEX

Main entries are given in **bold type**

Acknowledgments

Dorling Kindersley would specially like to thank Ian Graham for his significant contribution to this series. Thanks are also due to Fred Gill for checking the text and to Richard Bird for preparing the index. Commodore Business Machines (UK) Ltd kindly helped in the supply of equipment.

DK

Screen Shot

PROGRAMMING SERIES

The original and exciting new teach-yourself programming course for Commodore 64 owners.
Over 300 unique screen-shot photographs of program listings and programs in action – showing on the page exactly what appears on the screen.

Packed full of programming tips and techniques, reference charts and tables, and advice on how to get the most out of your Commodore 64.

CONTENTS INCLUDE

Setting up and starting off • Inside your computer • Screen layout and how to control it • Computer conversations • Programming with sprites • Animation • Special effects • Compiling a data bank • High-resolution programming • Curves and circles • 'Natural' graphics • Designing your own characters • Sprite animation, overlaps and collisions • Pies and slices • Bars and graphs • A guide to writing games • Shaping sound

CN 9384