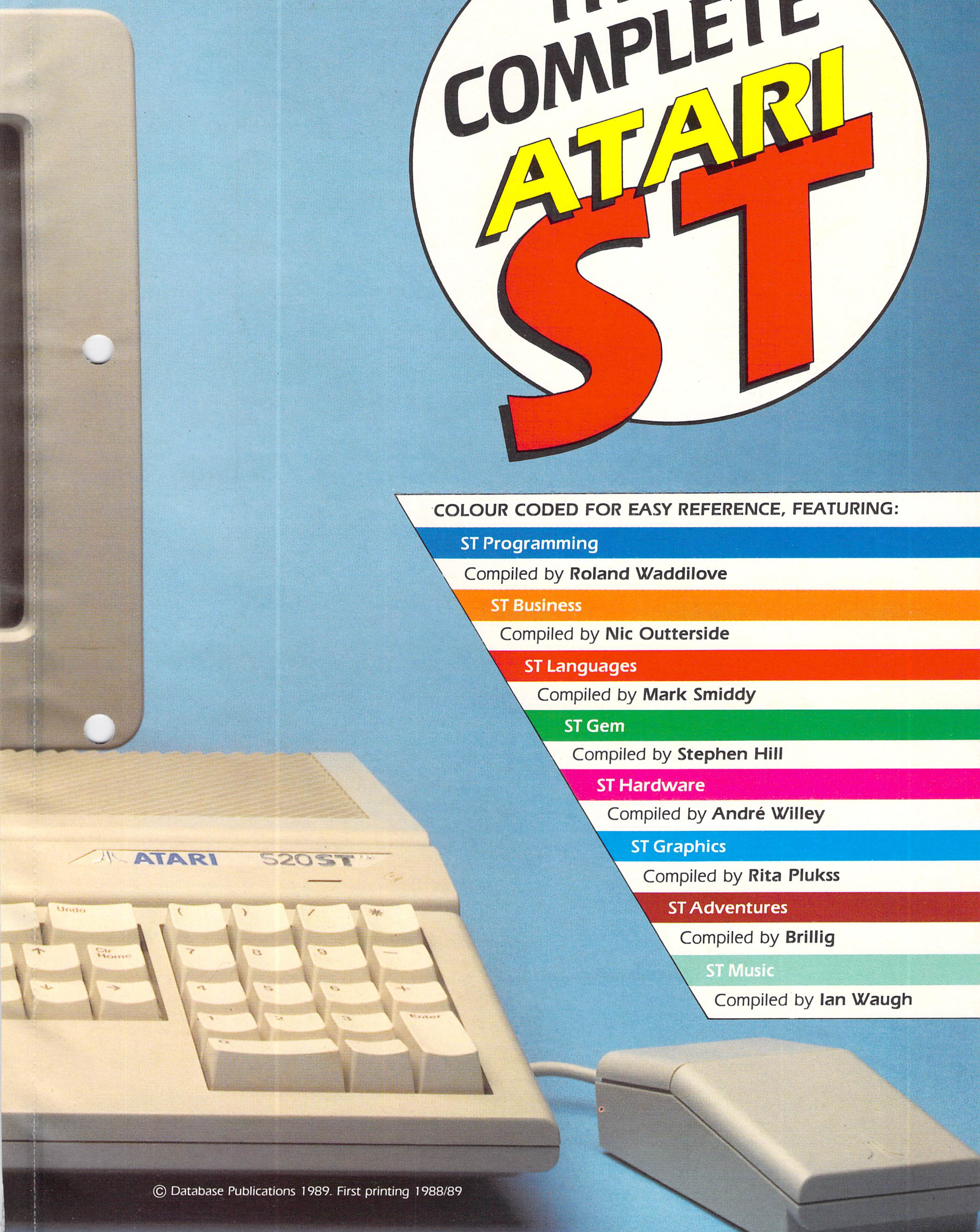
THE COMPLETE ATARI ST



COLOUR CODED FOR EASY REFERENCE, FEATURING:

ST Programming
Compiled by **Roland Waddilove**

ST Business
Compiled by **Nic Outterside**

ST Languages
Compiled by **Mark Smiddy**

ST Gem
Compiled by **Stephen Hill**

ST Hardware
Compiled by **André Willey**

ST Graphics
Compiled by **Rita Plukss**

ST Adventures
Compiled by **Brillig**

ST Music
Compiled by **Ian Waugh**

Index to The Complete Atari ST pull-out guide

Beginner's guide to adventures

LEAVING aside for one moment the keen ST games player, the word adventure will conjure up different things to different people. Travelling in a foreign country, meeting new people, doing something out of the ordinary or going somewhere unusual – all reflect most people's personal experience of an adventure.

However, we all know that adventures mean a lot more than, say, going for a holiday in Tunisia or having your first ever ride on a roller-coaster.

Which of us has ever had the chance to sail the high seas with infamous Captain Kidd and his bloodthirsty band? Sweated through the swamps and steam heat of the Amazon jungle in search of the lost tribe of Lali? Wrestled with magic spells and powerful potions to bring to book the evil Wizard of Wottsitt? Or attempted to stalk the illusive and deadly Phantom of the Fog through the grimy, cobbled streets of old London town?

Participation in such exotic adventures is beyond our reach, and we can only experience the excitement and danger through the medium of books, cinema, theatre and TV.

The main drawback to obtaining our thrills this way is that we are neither the focus of the action nor are we in control – we may only sit back and watch. The chills and spills can only be felt remotely – the danger threatens someone else, not us.

But thanks to the Atari ST and other home computers, we thrill-starved mortals may now experience daring do and danger a little more realistically. When you play an adventure game you are put at the very centre of events.

True, the comfort and protection of home and hearth is still there – and a good job, too – but at least you can now be in charge of the actions and fate of the central character instead of sitting back helplessly and hoping for the best.

Can you accept the challenge?

Now you're on your way . . .

THE documentation accompanying the program will normally set the scene and give some indication of the background to the adventure and mission.

The game proceeds by giving, in words, a description of the current location – a cave or throne room, for instance – the exits from the location, and any objects that are plainly visible.

Normally, all descriptions and messages generated by the program involve or are addressed to the player personally, like:

"The damp air of this gloomy chamber pervades your already-chilled bones. You can't help but notice that a huge cobra is slowly uncoiling its glistening length from around one of the two marble pillars, its eyes fixed unerringly on yours".

Play continues with you typing in a command which is followed by the computer's response.

Most adventures are not played in real-time so in effect time stands still while you are wondering what to do next.

Movement to another location is generally accomplished by leaving through one of the specified exits by typing in the appropriate command, such as GO EAST.

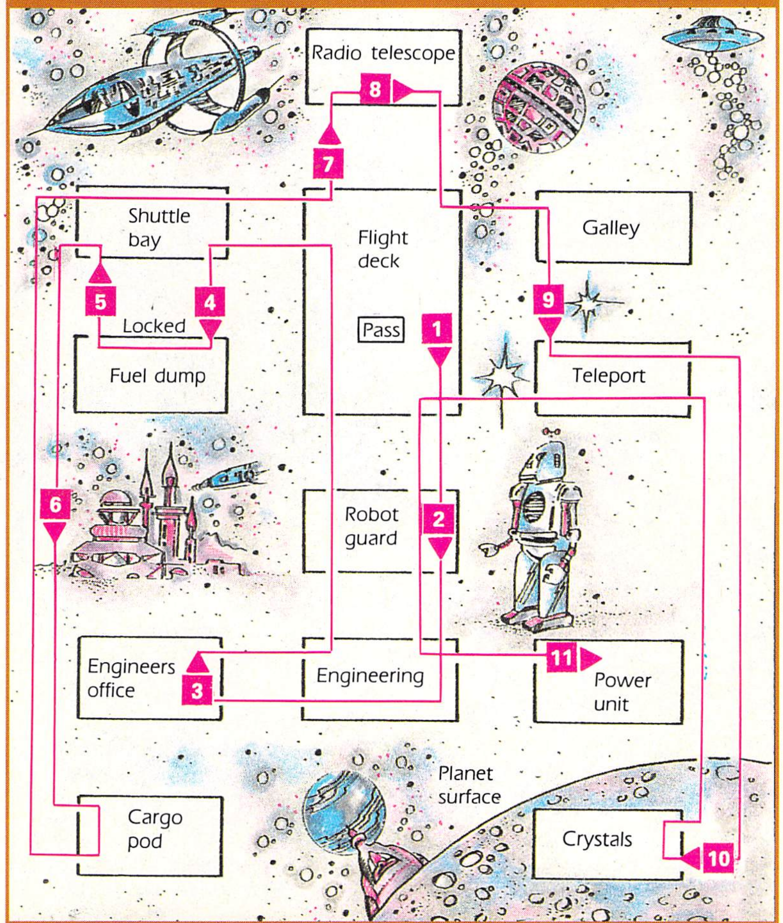
The program would then respond by displaying a description of the new location.

Again by keyboard commands, you can usually take and manipulate any objects found, but not necessarily all of them are vital to the successful completion of the mission. And so the adventure continues, with more locations, objects, challenges and events unfolding.

The finest adventures are not linear in construction – that is, you do not have to progress and solve all the puzzles in a single, pre-set pattern.

Instead you can take alternative routes to your goal and unravel many of the mysteries, within limits, in any order you wish.

The structure of an adventure



PROBLEM

You awake to find yourself alone on a deserted space ship. The flight deck is dead, apart from a sole message reading: "Replace crystals imminent". From your basic understanding of engineering you realise that the power crystals are lifeless

The ship is still operating on emergency power but time is limited. You must quickly find your way to the mines on the planet below you.

Only by replacing the crystals can you hope for escape.

SOLUTION

1 Take pass from flight deck.
2 Show pass to robot guard.
3 Take key from engineer's office.
4 Unlock fuel dump with key.
5 Refuel shuttle in bay.
6 Fly to cargo pod for satellite dish.
7 Connect dish to radio telescope.
8 Punch out planet coordinates.
9 Go to teleport and use coordinates.
10 Collect crystals and return.
11 Replace crystals in power unit.

Beginner's guide to adventures

The challenge

WITH two very important exceptions, playing a computer adventure is comparable to reading an exciting novel – not that we are trying to argue that playing an intelligent computer game is better for you than reading a good book.

What we wish to demonstrate in this section is that in a couple of specific areas, a well-written computer adventure can entertain, challenge and thrill you in ways that conventional stories can not.

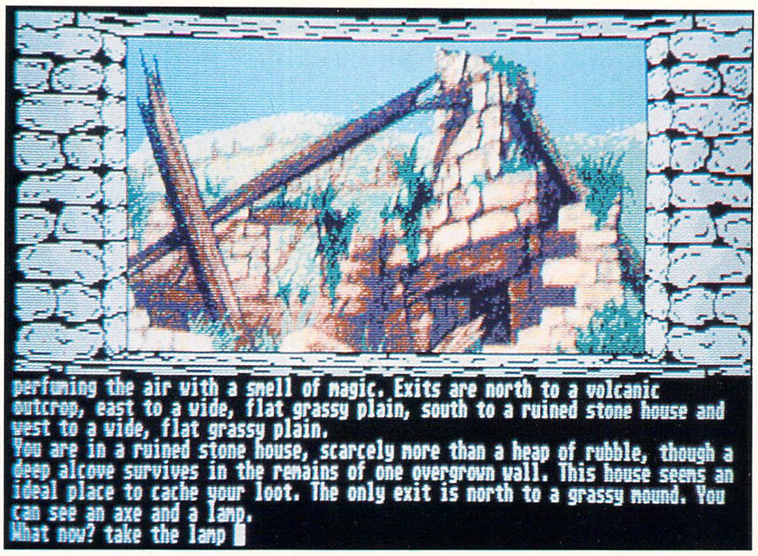
Firstly, in the computer novel, you are usually the central character and so what is happening to the star is in effect happening to you. Secondly, and perhaps more importantly, you can to a large degree control the progress and pattern of events and so, within certain limits, be master of your own destiny.

In a book everything happens in the same order every time. No matter how many times you re-read it the story, characters and events will always unfold in exactly the same way every time. Not so in a computer adventure.

Most games players have their own idea of what a computer adventure is, so before looking closely at the wide world of adventure programs let's simplify matters by defining such a program as a text-based story which allows the player to interact with it and affect its progress by means of typed commands.

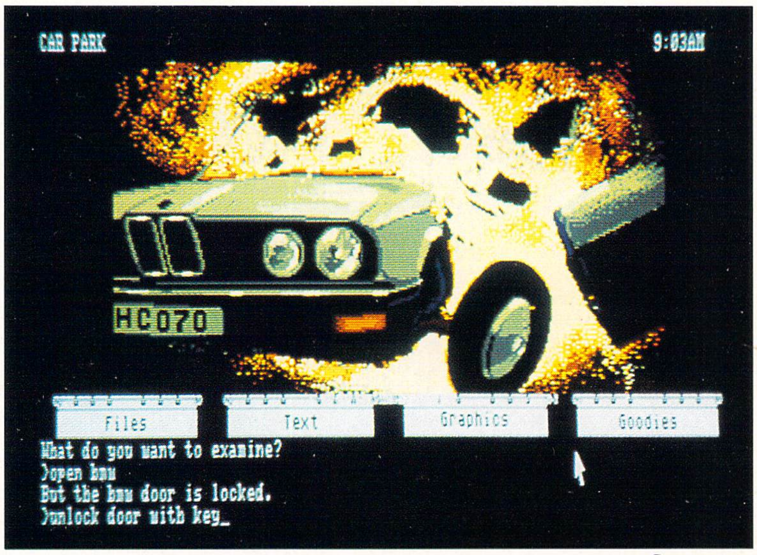
Although in this section I will follow the convention of referring to adventurers as males – because male players are in the majority – I realise there are many female adventurers. No male chauvinist piggery is intended.

An increasing number of adventure programs now have a female, or permit the choice of female or male, as the leading character.



```
perfuming the air with a smell of magic. Exits are north to a volcanic
outcrop, east to a wide, flat grassy plain, south to a ruined stone house and
west to a wide, flat grassy plain.
You are in a ruined stone house, scarcely more than a heap of rubble, though a
deep alcove survives in the remains of one overgrown wall. This house seems an
ideal place to cache your loot. The only exit is north to a grassy mound. You
can see an axe and a lamp.
What now? take the lamp
```

Time & Magik



```
CAR PARK                                          9:03AM

What do you want to examine?
)open bmw
But the bmw door is locked.
)unlock door with key_
```

Corruption

The puzzles

UNRESTRAINED progress is not always possible. This is where one of the adventurer's chief delights – the puzzle – comes in.

Taking a very simple example, you may find yourself in a room where one of the exits is a locked door.

You can't open it unless you have the correct key. But where is it? Perhaps you failed to notice that the rug in a nearby bedroom could be picked up. If you had, you would have discovered a shiny, brass key concealed beneath it.

Setting the scene

ADVENTURES can be set in the past, present or future, or even outside time itself. The environment might be anything from a ghost town to an alien planet, from a great underground empire to a Hollywood mansion, from a mythical world where magic and sorcery prevails to a futuristic land where science and technology are masters.

The only limit to the setting of an adventure is the author's imagination. All the best adventures have a consistent, rational theme and an objective for you to aim at.

Your prime task might be to solve a crime, thwart an invasion of Earth, find a group of treasures, rescue a hostage, become a better person or construct a machine – but not all in the same adventure.

The list of possibilities is endless. I have even played an adventure where the main aim was to collect fluff.

The parser

THE adventurer's keyboard commands are interpreted by the program's parser, a group of routines which break down input and attempt to make sense of it.

Associated with the parser is the vocabulary, a built-in store of verbs, nouns, adjectives and so on, against which each word of your command is compared.

The program can only comprehend words contained in its vocabulary so the larger it is, the more likely the program is to understand your commands.

Once your input has been interpreted, the program determines whether the command makes sense. Take DRINK THE HOUSE – although all the words may be recognised by the program, the command is clearly nonsense.

If a command does make sense the program will check whether the circumstances are appropriate for it, and, if so, what the consequences are.

For instance, if you were in a dark dungeon and typed LIGHT THE MATCH but didn't actually have one, the program would more than likely politely tell you that it would help if you really had a match.

Assuming you did, the program might display a message that said you could now see the hitherto hidden spiked pit into which you were about to fall.

On the other hand, if there was a serious leakage of gas you had failed to detect in the dungeon, then the result of lighting the match might be the program telling you that you had just been blown to pieces.

Creating the right atmosphere

ONE of the chief pleasures of adventuring is that of escaping into another world, leaving behind everyday life and being so diverted and engrossed by the skills of the author that you practically forget it is only a game, and really become absorbed in the story and role assigned to you.

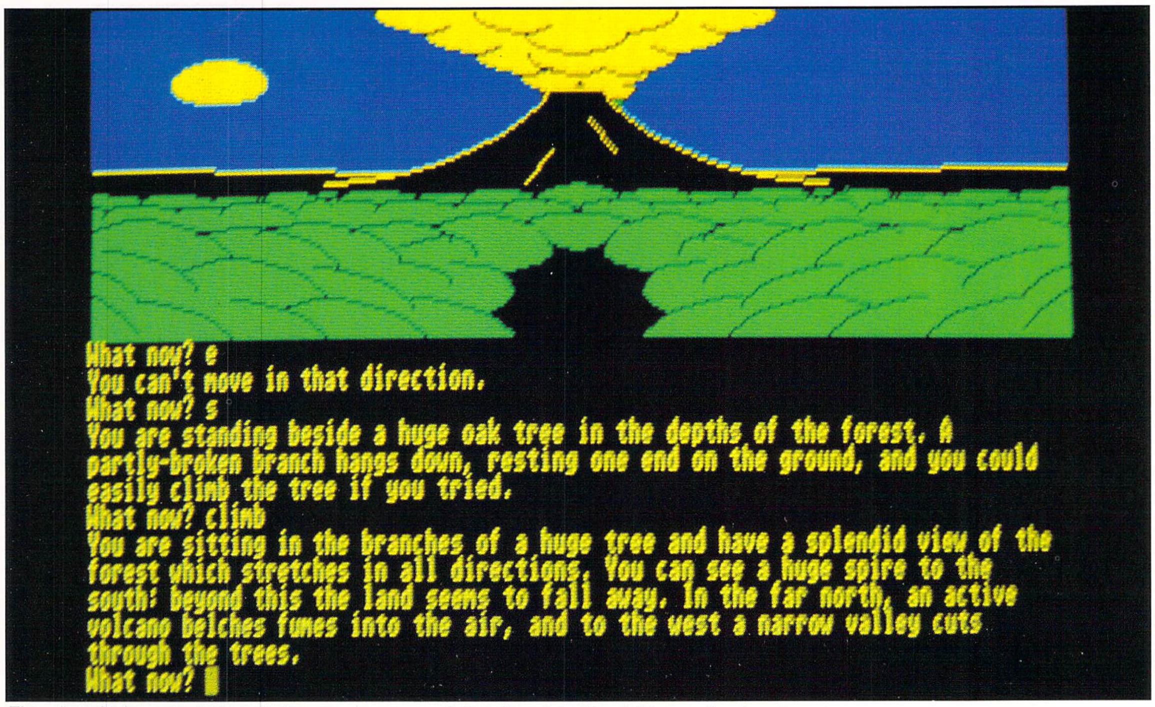
For such successful escapism the adventure must generate a suitably gripping atmosphere and hang together logically, even if the logic happens to be that of an entirely different culture.

The main device for creating such an atmosphere is the use of vivid and fulsome descriptions in the story. The more detailed and literate the narrative, the more the adventure engages the player's heart and mind.

The poorer adventures offer skimpy and banal descriptions such as "You are in a gloomy cave. There is a knife here. Exits are east and down". Not terribly exciting. Such works are the product of impoverished or lazy imaginations and are rarely worthy of your attention.

They try to compensate by offering a greater number of locations, characters and puzzles, but to my mind such adjustments are scant reward.

Unless a strong atmosphere is created much of the enjoyment and raison d'être of an adventure is lost.



```
What now? e
You can't move in that direction.
What now? s
You are standing beside a huge oak tree in the depths of the forest. A
partly-broken branch hangs down, resting one end on the ground, and you could
easily climb the tree if you tried.
What now? climb
You are sitting in the branches of a huge tree and have a splendid view of the
forest which stretches in all directions. You can see a huge spire to the
south: beyond this the land seems to fall away. In the far north, an active
volcano belches fumes into the air, and to the west a narrow valley cuts
through the trees.
What now?
```

The descriptions in Colossal Cave are some of the finest in the history of adventuredom

Colossal Cave

THE better adventures provide rich and finely honed text, rather like a good book. One of the best – and it so happens, the first – adventures ever written is the famous Colossal Cave by Crowther and Woods. The descriptions in this classic fantasy are some of the finest in the history of adventuredom.

Although it was originally written for a mainframe computer, you can nevertheless obtain an almost identical version for your ST. By the use of some clever text compression techniques, Level 9 managed to get it to run on a home computer. Colossal Adventure, as the adaptation is called, is currently marketed by Rainbird as part of a trilogy called Jewels of Darkness.

To illustrate the excellence of this and to show the sort of standard you should be looking for in a quality adventure, compare the following excerpts from Colossal Cave with the example given above:

"You are on the edge of a breathtaking view. Far below you is an active volcano from which great gouts of molten lava come surging out, cascading back down into the depths. The glowing rock fills the farthest reaches of the cavern with a blood-red glare, giving everything an eerie, macabre appearance.

"The air is filled with flickering sparks of ash and a heavy smell of brimstone. The walls are hot to the touch, and the thundering of the volcano drowns out all other sounds. Embedded in the jagged roof far overhead are myriad twisted formations composed of pure white alabaster which scatter the murky light into sinister apparitions upon the walls.

"To one side is a deep gorge filled with a bizarre chaos of tortured rock which seems to have been crafted by the devil himself.

"An immense river of fire crashes out from the depths of the volcano, burns its way through the gorge, and plummets into a bottomless pit far off to your left.

"To the right, an immense geyser of blistering steam erupts continuously from a barren island in the centre of a sulphurous lake, which bubbles ominously. The far right wall is aflame with an incandescence of its own which lends an additional infernal splendour to the already hellish scene. A dark, foreboding passage leads to the south".

Now is that a description or is that a description? In fact, Colossal Cave is the yardstick by which all other adventures should be measured.

Adventure topography

NARRATIVE quality is not the only element that goes to make up a good adventure. Topography is significant too, and must be consistent both with the rest of the contents of the adventure and within itself.

For instance, if you leave a dungeon via the North door and arrive in a murky chamber, you should expect to find yourself back in the dungeon if you exit via the South door.

Similarly, if you have scrambled down a slippery slope it does not necessarily follow that you should be able to go up it again without difficulty. Perhaps a rope tied round a boulder at the top of the incline would assist.

As another example, if you've cut a hole in a sheik's tent in order to escape, you should expect to find it still there if you return. If it had been repaired you would expect some explanation to be proffered if credibility is to be maintained.

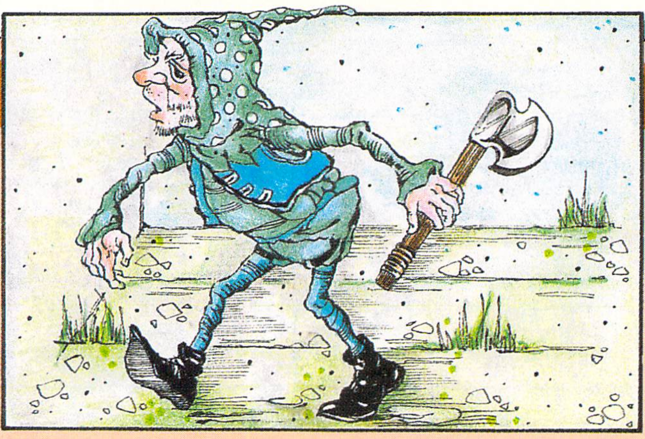
Logic and consistency in the topography is critical if the atmosphere is to

Creating the right atmosphere

Keeping up interest

ANOTHER element of a good adventure is the inclusion of plenty of interesting incidents and events. A static adventure, no matter how vibrant the text or how imaginative the land, can be a bore.

This further example from Colossal Cave should give you some idea of what I mean: "A little dwarf just walked round the corner, saw you, threw a little axe at you which missed, cursed and ran away".



be sustained and your enjoyment is not to be marred by illogicalities.

As there are usually a large number of connecting locations a simple methodology is to draw each location as a small uniquely labelled box with the exits depicted as straight lines emanating from the appropriate points.

For instance, North would be the top of the box, South the bottom, Northwest the top left corner and so on. Up and down – stairs and hill paths are pertinent examples – can be shown by a zigzag line at the top or bottom of the box respectively.

By linking these boxes together as you progress through the adventure you will be able to retrace your steps whenever you wish.

A maze in adventures

MAPPING works well until you hit one of the banes of an adventurer's life – the maze. These used to be the staple diet of adventures and you would be hard put to find one that didn't have one in some form or another. These days they are less common, mostly because all the variations have been rung out of them, and because many experienced adventurers find them too tedious.

The basic idea was to have a series of linked locations, every one of which had the same description. Relying on the position of the exits to differentiate between one location and another was not a lot of good either since many were identical even though they in fact were separate places.

Simply taking pot luck and moving from one spot to another in the hope that you might stumble on the exit was a sure way to get nowhere fast because for all you knew you could have been going round in circles.

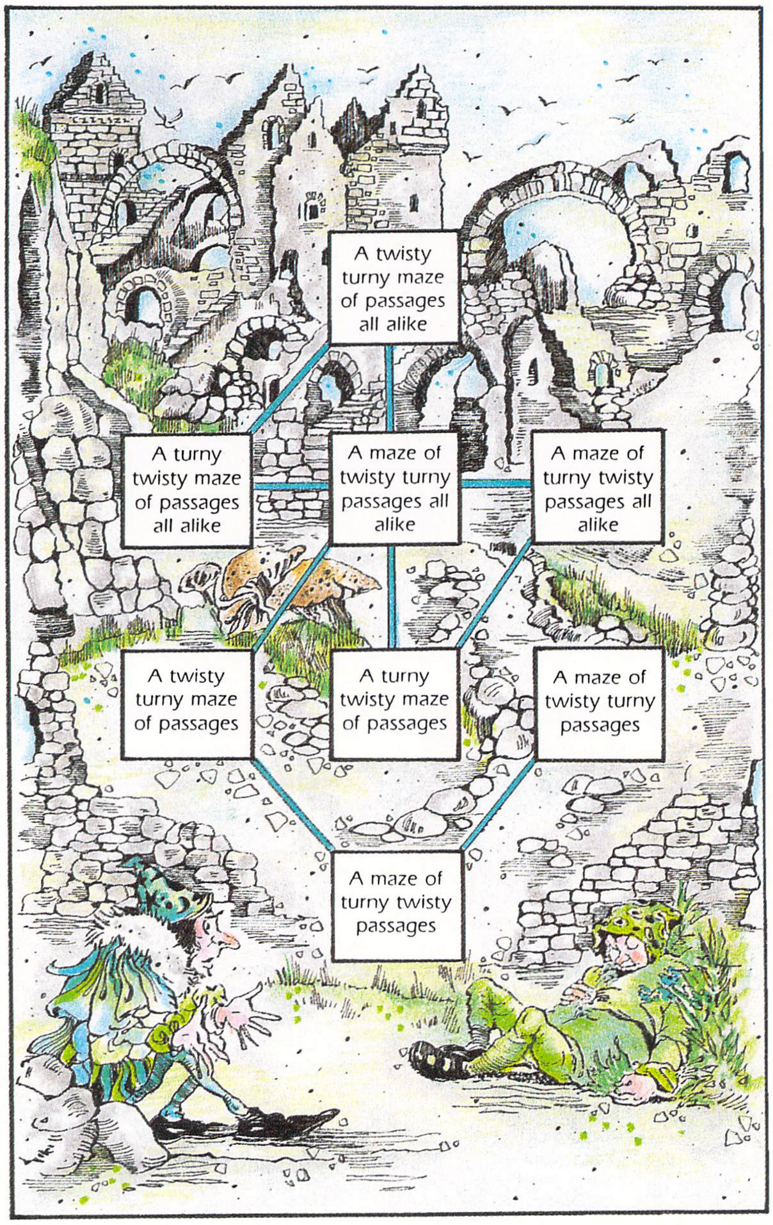
Colossal Cave had a maze where each room had a small, easily-missed variation in its description. Each was thus uniquely and fairly described, but you had to be on your toes to spot it.

The traditional method of cracking mazes is to grab as many objects as you can before entering, then drop one in each location. Even though the location description is still the same, the presence of a dropped object uniquely identifies it – adventures always tell you what is visible. It usually takes several forays into a maze to map it as there is a limit to how many objects you can carry.

Some of the more fiendish adventure writers use ploys to thwart this system of maze solving. One had a thick mist on the floor so anything dropped was lost to sight. The solution was to carry a huge fan.

Another had a small dragon that ran in, gobbled up the dropped item, then ran out again, while another adventure would not allow you to carry anything except a lamp. The secret was to switch it off – in the dark, a luminous arrow pointed to the exit.

Yet another had a character who would move into a location you had left, pick up the object and place it in another room just to confuse you.


A twisty turny maze of passages all alike
A turny twisty maze of passages all alike
A maze of twisty turny passages all alike
A maze of turny twisty passages all alike
A twisty turny maze of passages
A turny twisty maze of passages
A maze of twisty turny passages
A maze of turny twisty passages
A typical maze

Deciphering riddles

EVEN though it might have a gripping plot, marvellous characters, a sophisticated and intelligent command analyser and a vocabulary as big as the Concise Oxford Dictionary, for most people an adventure just wouldn't be an adventure without one important ingredient – puzzles.

They come in all forms and guises and sometimes may be so heavily veiled that you may not even recognise them as puzzles at all. In the best adventures, the problems are an intrinsic part of the plot and are not merely tacked on as intellectual appendages.

While puzzles may be interlinked, you should not be compelled to solve them all in a strictly linear fashion.

Nothing can be more frustrating than getting stuck near the beginning of an adventure where the failure to solve that specific teaser prevents you from making any further progress.

One particular example of such a puzzle occurs in Hollywood Hijinx in which you start off outside a mansion. Unless you can find a way into the house you're not going to make any headway with the game.

The solution, which to be fair is pointed at in an obscure rhyme supplied with the game's documentation, involves turning a statue several times so that at each turn it faces in a set direction. Get the sequence wrong and nothing happens. Follow it correctly and the front door of the house automatically unlocks at the final turn of the statue.

This tough nut of a puzzle had many adventurers tearing their hair out in frustration. What made it worse was the gigantic maze in the grounds of the mansion, leading you to believe there was something in it which would assist entrance to the house.

There wasn't.

A really tough puzzle is legitimate but it should never be a bar to progress near the opening of the adventure.

The Babel fish problem

PUZZLES should be fair, logical, mentally challenging and relevant to the development of the adventure. They may range from the simple, such as sliding a newspaper under a door to catch a key falling from the lock on the other side, to the complex and multi-layered, such as the famous Babel fish mind-bender in Hitch Hiker's Guide To The Galaxy.

In this puzzle, the idea is to obtain a small fish from a dispenser situated on board a Vogon spacecraft. When placed in the ear, it enables the hearer to understand alien tongues – and in particular, the speech of the captain of the Vogon Destructor fleet. Failure to translate the Vogon's speech will cause you no end of trouble later in the game, so possession of the fish is vital.

This may all sound pretty daft, but anyone who has heard the radio plays, read the books or seen the TV series will be familiar with the unique humour of Douglas Adams which pervades in this superb adventure.

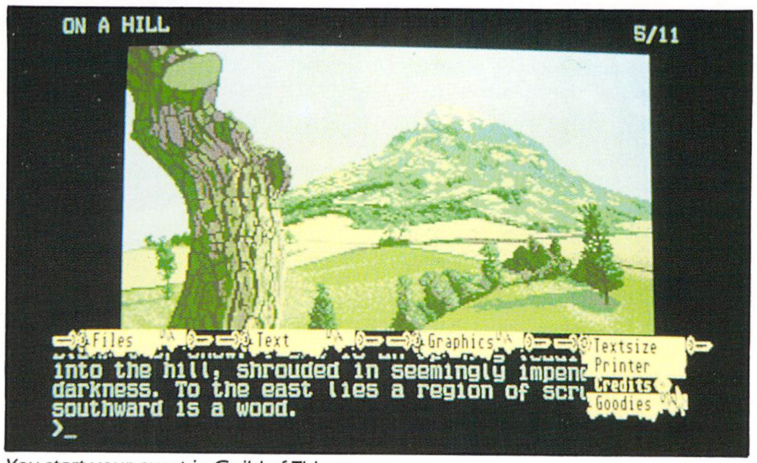
Pressing the dispenser's button causes the fish to shoot across the room at shoulder height and disappear through a hole in the wall. The obvious solution, therefore, seems to be to find some way of blocking up the hole. Careful examination of the wall reveals a small hook above the hole, just right for hanging a dressing gown on and thus covering it.

Easy – except that this time the button is pressed, the fish hits the gown, slides down inside one of the sleeves and drops through a drain in the floor. Knotting the sleeves of the dressing gown doesn't help – next time the Babel fish slides down the outside of the sleeve.

Covering the grate with a towel seems to be a good idea – only now, as the fish lies wriggling on the towel, a tiny cleaner-robot appears, sweeps it up and exits through a small panel at the foot of one of the walls. It's almost as if the author of the adventure is reading your thoughts and staying one step ahead all the time.

The panel can be obstructed by placing a satchel in front of it – except that when the robot crashes into the satchel, the impact causes the fish to be thrown into the air where it is caught by a flying, junk-seeking robot. What to do now? The puzzle seems to have as many layers as an onion.

The word "junk" turns out to be a vital clue. When a pile of junk mail – which you may well have left behind on Earth, believing it to be useless – is placed on top of the satchel, next time the sequence takes place, the mail flies into the air along with the fish. The aerial robot is so distracted by all the litter flying around, it misses the fish, which falls straight into the player's ear – at last!


```
ON A HILL                                  5/11
into the hill, shrouded in seemingly impene
darkness. To the east lies a region of scru
southward is a wood.
>_
```

You start your quest in Guild of Thieves

Hints and tips

JUST as in the example of the "junk" clue above, sufficient hints and information should be given within the adventure itself to point you at the correct solution without handing it to you on a plate. Ideally, the puzzles should be capable of being solved in more than one way and the solution should not rely on specialist knowledge.

Further examples of built-in hints, albeit in a more subtle manner, occur in Guild of Thieves and Mordon's Quest. In Guild, one puzzle involves opening an opaque case by manipulating a set of dice. The main problem is knowing how many spots should be showing on the top face of each die when it is placed in a slot. Your map provides a big clue – the arrangement of rooms surrounding the opaque case resembles the five-spot face of a die.

Similarly, in Mordon's Quest a password is required at a certain stage. Careful examination of the map reveals that the location arrangement of this particular region resembles a frog – and that turns out to be the password.

When designing a puzzle, it is reasonable for an adventure author to expect a certain level of general knowledge on your part. For instance, turning to Guild of Thieves once again, a room full of coloured squares has to be crossed, each square being stepped on in the correct sequence – one false move means instant obliteration.

A sign over the entrance reads WOBNIAR – fairly meaningless unless the penny drops that the word is actually RAINBOW spelled backwards. So providing you remember what you were taught at school about the order of colours (Richard Of York Goes Back

Deciphering riddles

In Vain was how my teacher drilled it into me, the initial letters standing for each colour), cracking the problem becomes easy.

Red herrings and other objects

MANY puzzles involve the manipulation of an object, the normal purpose of which may disguise the fact that it can be used in another way. It is therefore vital that you collect as many items as possible as you go on your travels.

Some objects may turn out to be of no relevance whatsoever and some may be capable of being used in different ways to solve more than one puzzle. Until the adventure is completed a wise adventurer will discard nothing, just in case.

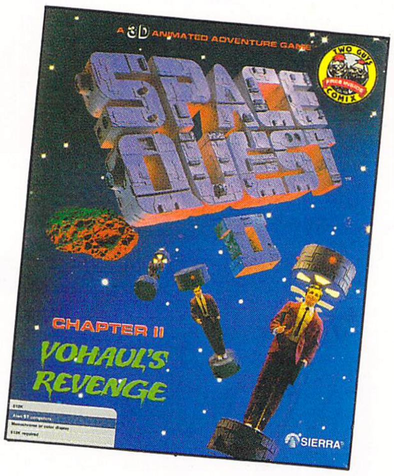
One of the problems with amassing large numbers of objects is that many adventures set a limit to how much can be carried at any one time. A golden rule, therefore, is to carry only those things that look to be of immediate relevance – like a key, food, shovel, lamp, and so on – and store the rest at as central a site as possible for later easy access.

Not all the objects turn out to be useful. In Scott Adams' Pirate Adventure, the adventurer will come across a mongoose, and knowing how such creatures are skilled at killing snakes, will spend much time carting it around.

Near the end of the adventure, an uncrossable pit full of deadly snakes is found. You naturally assume that the mongoose will solve the problem and the snakes kill the mongoose.

It transpires that it wasn't a mongoose after all but a squirrel, and another means of disposing of the snakes has to be found. Rather a dirty trick on the part of the author, but no adventurer worthy of the name should take everything at face value in these games.

Another such crafty trick is played in Guild of Thieves. In this a certain path is blocked by a barred gate which cannot be climbed or unlocked. The answer is simply to break the bars –



they turn out to be made of polystyrene.

In Colossal Cave a fierce dragon sits on a rug. If you attempt to fight it, the program responds: "What, with your bare hands?" and most people at this point take the hint and look for other ways of overcoming the beast.

However, for anyone foolish enough to enter "yes" in answer to the question, the program goes on to describe how you manage to kill it with your bare hands, and adds that anything is possible in an adventure.

Lateral thinking

IN many cases lateral thinking is required to solve a particular puzzle. In Space Quest II you are forced to climb down a ladder leading deep into a pitch-black shaft and thence into a labyrinth of tunnels.

A light is essential, but the only source you have access to is a glowing gem. As both hands are needed to hold the ladder, how can you hold the gem so that its light shines out? The answer seems obvious – once you know it. The gem has to be held in your teeth.

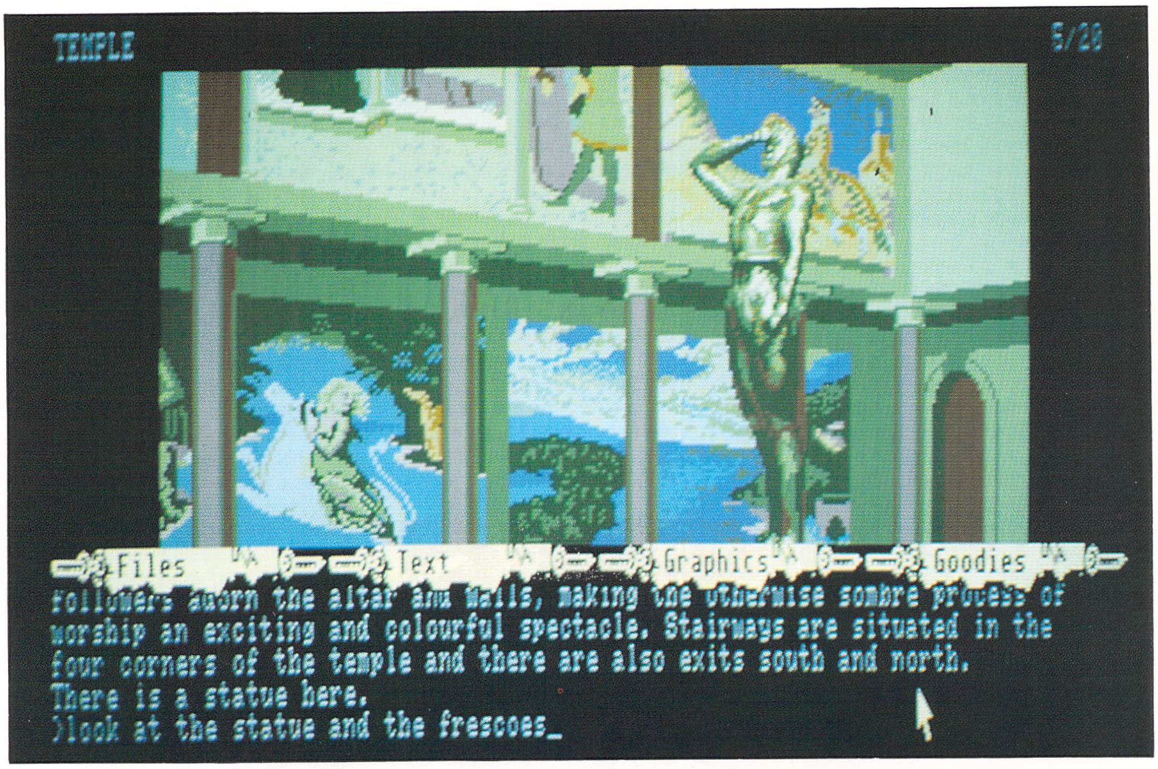
Some items vital to completing an adventure may come in a less expected form. For instance, a source of light is often required to explore dark regions but an oil lamp may not always be readily available.

In that situation you would be well advised to keep a sharp look out for such things as phosphorescent moss, glowing jewels, candles, bundles of rags, flint and certainly matches.

In one adventure you need to have a light source after swimming underwater – the problem is that the matches always become wet and useless during the swim. The solution is to waterproof them by coating them in candle wax before taking the plunge.

All adventurers have been baffled and bamboozled by at least one puzzle in their lives, but any frustration is more than compensated for by the pleasure and sense of achievement when a real brain-teaser at last yields to your probings.

Without puzzles, adventures would be the poorer – and so would our enjoyment.



```
followers adorn the altar and walls, making the otherwise sombre process of
worship an exciting and colourful spectacle. Stairways are situated in the
four corners of the temple and there are also exits south and north.
There is a statue here.
>look at the statue and the frescoes_
```

The Temple in Guild of Thieves poses problems of its own

It's all a plot

A STRONG atmosphere, attention to detail, interesting incident, characterisation, a comprehensive vocabulary and a flexible command parser are all important features that go to make up a good adventure. But there is one further element that is of fundamental significance to an adventure's success or failure – the plot.

As with novels, the storyline in an adventure can be serious or funny, real or fantastic, original or traditional. Many people are under the misapprehension that adventures are always about trolls, goblins, wizards and assorted Tolkien-cloned creatures.

It is true that many adventures do indeed have a sword and sorcery theme, but that is by no means an accurate representation of the world of adventures, as we shall see.

Thrillers

CRIME, and in particular the whodunnit stream of literature, has long been a favourite source for adventure plots. Deadline, from the Infocom stable, sets you in a race against the clock to uncover the facts behind mysterious death of an industrialist.

As the chief of detectives in a fictitious American town, you have been asked by the deceased's lawyer to investigate the death – by apparent drug overdose – of his client. It seems that the victim had called his lawyer only three days earlier for the purpose of setting in motion the necessary procedures to change his will. His unexpected and seeming suicide has now prevented any alteration to the will.

There are two especially interesting things about Deadline. The first is that you don't not know at the start whether there really has been any foul play. The second is that in addition to snooping around for clues, you can actually interrogate all the characters in the adventure.

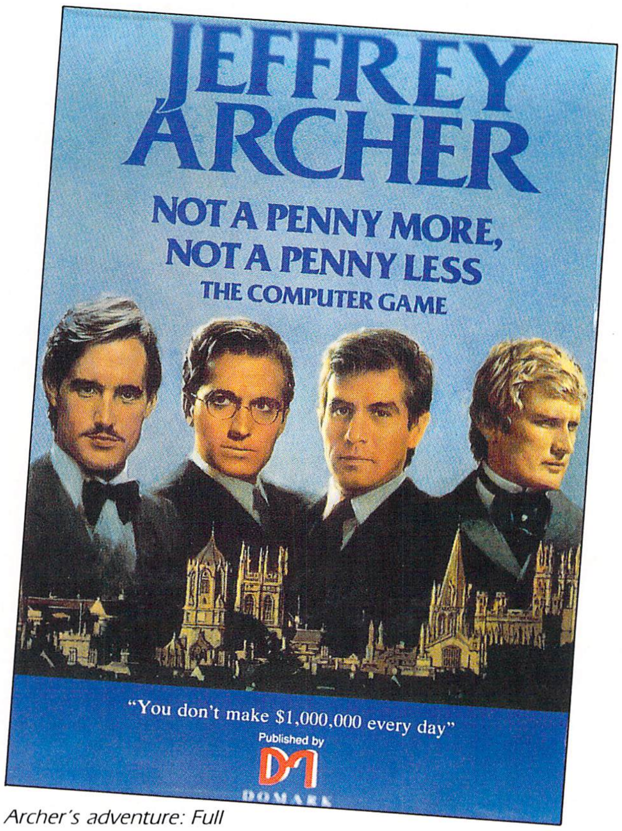
The style, timeliness, manner and content of these interrogations determines the responses of the characters – and the eventual outcome of the story. This really is interactive fiction at its best.

Perry Mason is one of the most famous characters in crime fiction and in Telarium's The Case of the Mandarin Murder, you are given the opportunity to play the role of Perry in one of his most difficult cases.

Laura Knapp is about to be divorced by her famous restauranteur husband, Victor, and engages Perry to take on the lawsuit. Barely 12 hours later, Victor has been murdered – with all the evidence pointing slap-bang straight at Laura. But she swears she is innocent.

As Perry, you are required to investigate the scene of the crime, use secretary Della Street and private eye Paul Drake to dig up evidence and get the background on any other suspects. You must then bring the case to court and conduct the defence of your client.

As well as having a juicy plot, this adventure is unique in allowing you to do most of the things a court-room lawyer would do – cross-examine witnesses, introduce evidence, consult privately with the judge, and use every trick in the book to expose the true murderer and get your client off the hook.


JEFFREY ARCHER
NOT A PENNY MORE, NOT A PENNY LESS
THE COMPUTER GAME
"You don't make $1,000,000 every day"
Published by
DOMARK

Archer's adventure: Full of shady double dealing

Another development of the same theme is Domark's wobbly adaption of Jeffrey Archer's Not a Penny More, Not a Penny Less. In this you are cast as Stephen Bradley, the American academic working at Magdelan College, Oxford.

Your task is to recover the money you lost by investing in an oil company floated by that rogue, Harvey Metcalfe. The plot is full of shady double dealing and corruption as you struggle for good to conquer villainy. It is a thriller in the true sense of the word. Despite poor parser and lack lustre game development the tension is continued to the ''Big Bang'' climax at the end.

Science fiction

TURNING from crime to science fiction, Ray Bradbury's classic story Fahrenheit 451 – its name derives from the temperature at which books will burn – has been turned into an adventure.

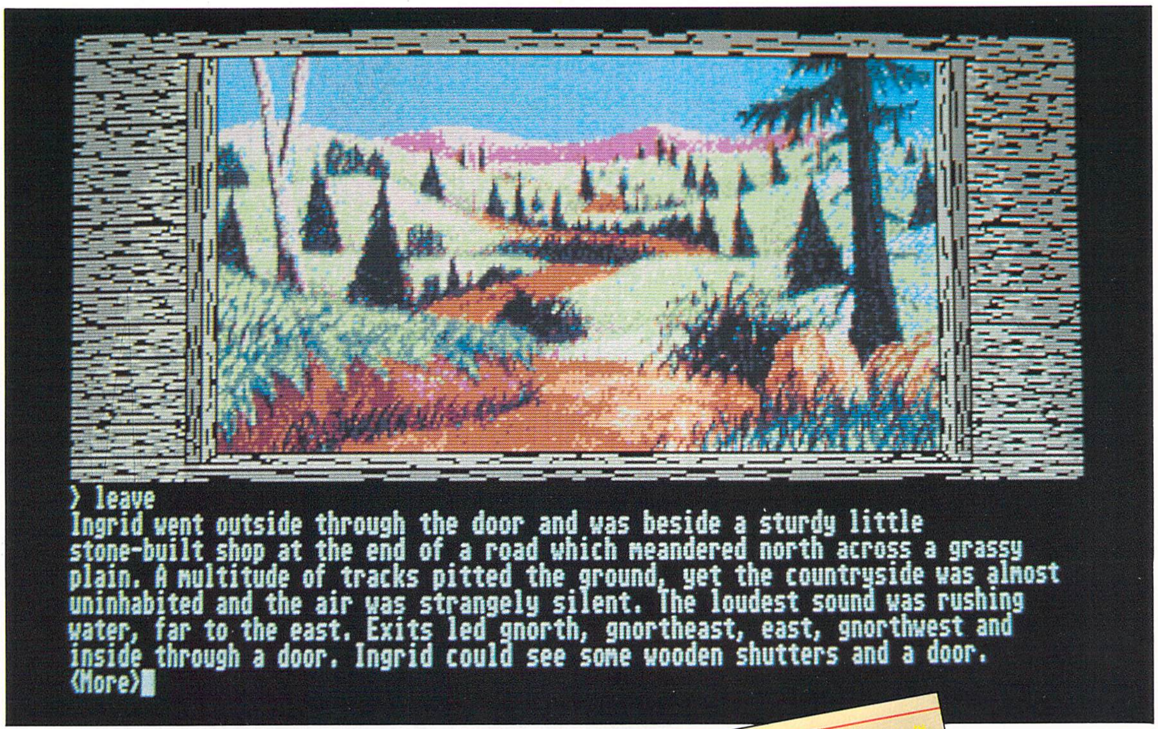
The tale is set in a future where it is illegal to possess books, and a fireman comes, not to save houses, but to burn them and the books inside, where rebels turn themselves into living books by memorising literary works word for word.

In this adventure you take on the role of rebel fireman Guy Montag who is on the run from the state and is seeking to join the underground resistance intent on restoring freedom to the world. Fahrenheit 451 combines an unusual and gripping plot with literate prose and exciting incidents, coupled with the uncertainty of trying to ascertain who are your enemies and who your friends.

One of the most original and imaginative of science fiction adventures is Infocom's Suspended, in which all your actions are carried out through six highly individual robots. Having awoken after many years from a state of cryogenic suspension, you discover that the planetary control systems are all going to pot. You are unable to move but must use the robots as extensions of yourself.

Iris is a visual robot with limited mobility while Auda has only aural capabilities. Sensa is a mixture of

It's all a plot



```
> leave
Ingrid went outside through the door and was beside a sturdy little
stone-built shop at the end of a road which meandered north across a grassy
plain. A multitude of tracks pitted the ground, yet the countryside was almost
uninhabited and the air was strangely silent. The loudest sound was rushing
water, far to the east. Exits led gnorth, gnortheast, east, gnorthwest and
inside through a door. Ingrid could see some wooden shutters and a door.
<More>
```

Gnome Ranger: A witty tale from Level 9

sensory apparatus while Waldo is an industrious robot, built chiefly to handle and manipulate objects.

Acting as an interfacing device between you and a massive databank, known as the Central Library Core, is Whiz. However, most peculiar of all the robots is Poet, who has powers of touch but can only describe what he perceives in the most florid of prose. While what he says is always accurate, Poet's use of the English language is bewildering, to say the least!

Carry on laughing

AN interesting combination of science fiction with humour is Infocom's The Hitch Hiker's Guide To The Galaxy. This is a completely zany and unique adventure based on Douglas Adams' famous radio series/book/TV series/ play/record of the same name.

The two Activision Space Quest adventures (I and II) are similarly funny and have the addition of animated graphics. Space Quest II has the unlikely scenario of you as a space janitor attempting to save the world from an invasion of mutant insurance salesmen!

Staying with humour, Level 9's Gnome Ranger is a witty tale about the mishaps of a bossy young gnome called Ingrid. Equally funny is Infocom's Wishbringer which offers a new and delightful approach to the traditional good and evil fantasy genre.

Rainbird's Guild of Thieves and Jinxter successfully manage to blend a sense of fun with a good story line and testing problems.

Leisure Suit Larry in The Land of the Lounge Lizards is definitely in the risque department of humour. It is a three dimensional animated graphic adventure full of inuendoes and sleeze as you play the feature role of Larry.

STATIONFALL
FLOYD IS BACK IN THE BOFFOID SEQUEL TO PLANETFALL
GET PLANETFALL FOR ONLY $14.95 OFFER ENCLOSED!
Leisure Suit Larry in
THE LAND OF THE LOUNGE LIZARDS
ANIMATED ADVENTURE GAME
SIERRA

Adventures cover far more than witches, wizards and warlocks

Turned 40 and a real footloose and fancy-free Del-boy character your aim is to burn both ends of the candle for one whole night. This involves gambling, drinking, dancing, girlies and overcoming your jerkisms.

The adventure is adult, very American but also very funny, and not a goblin in sight!

Finally, and once again from the illustrious Infocom, comes Stationfall which, while being extremely funny, novel and challenging, has perhaps the saddest ending of any adventure. The brilliant mixture of humour, imagination and pathos in a gripping plot makes Stationfall an adventure that will stay in your mind long after you've finished it.

I hope you can see from this brief overview that adventure plots cover far more than witches, wizards and warlocks.

Whatever your taste in stories and themes, you can be sure that there's every chance that somewhere, sometime, somebody has written an adventure that perfectly fills your personal niche.

Utilising objects

THERE are four golden rules of adventuring:
- **Make a map.**
- **Examine and search everything.**
- **Save your position at regular intervals.**
- **Collect all the objects you come across.**

It is with the last of this quartet of tenets that I shall concern myself in the following paragraphs. No matter how mundane or how obscure an object may appear to be at first sight, it is usually there for a specific purpose. Although the particular reason for its presence may not be at all apparent when you first encounter it, you can be fairly sure that somewhere, at some time, it will prove to be vital to further progress and the ultimate success of your quest.

THE cardinal rule is: Pick up every item that can be carried and take it with you. To be sure, some adventure authors do strew a number of red herring objects in their world, but since the adventurer will not know for certain until the end of the game whether or not an item has served any useful purpose, the only safe course is to gather it up. A wait and see approach is the best advice.

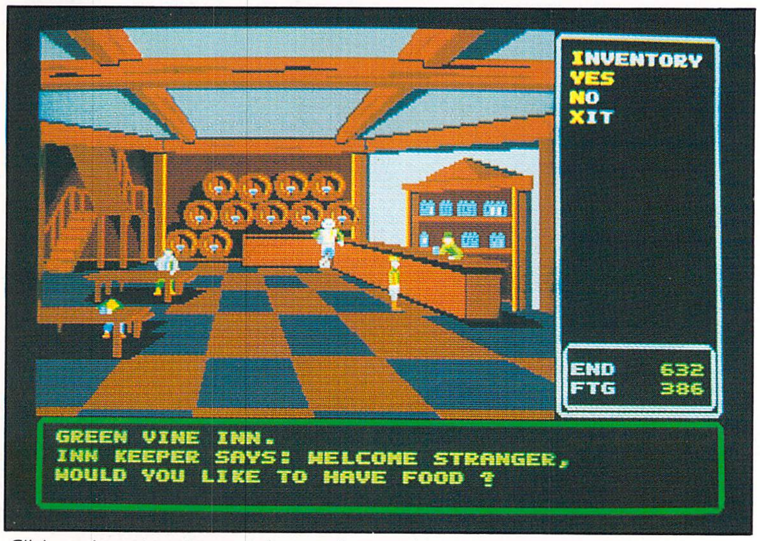
Most adventures put some form of restriction on the number, weight or type of objects that can be transported at any one time. If there is a knapsack, satchel, bag or any other form of container, it is as well to store as many objects in it as possible since the normal run of adventures allows you to carry more that way. Once a limit has been reached, some things will inevitably have to be left behind.

At this stage, it is often a good idea to separate what appear to be crucial items or those that have a seemingly obvious use from the rest. The essential objects can be carried while the surplus can be stored together, ideally at some easily reached, central location.

It doesn't always make sense to leave behind an object simply because you've already used it once – a crafty author may have designed the adventure so that there is a requirement for you to use a particular object to solve more than one puzzle.

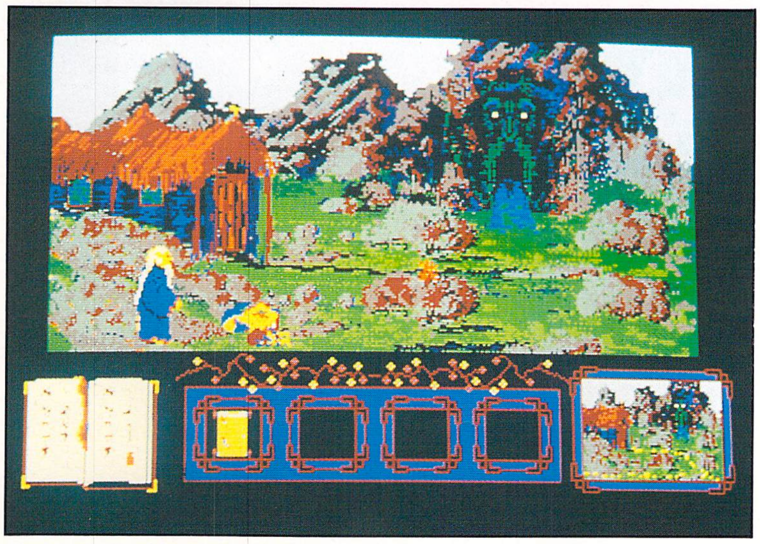
So far as is foreseeable, make sure you can always get back to the storage location. For instance, it's no use dumping some of your objects on a shore, sailing off in a boat to an island and then, once there, chopping up your boat to make a fire if there's no alternative way back to your repository.

Take care that nothing can get at your deserted objects while you're away. Thieves may be lurking nearby so try to secure or lock up your possessions if you can. Be on guard against acts of nature – in the previous example, it's possible that the author may have programmed the tide to come in and wash away your belongings before you have a chance to retrieve them.



GREEN VINE INN.
INN KEEPER SAYS: WELCOME STRANGER,
WOULD YOU LIKE TO HAVE FOOD ?

Click on Inventory to see what you're carrying



The contents of four pockets shown as icons

Search and discover

DESKS, cupboards, closets, sacks, vases – anything that is capable of containing something else – should always be searched since their role may be that of concealment. In that context, much less obvious containers such as grandfather clocks, ovens, radios, cornflake packets, golf bags, and the pockets of snooker tables should all be scrutinised.

Book shelves are fair game, too. The very act of taking down a book from a library shelf may cause a secret passage to open.

All books should be opened and read. The two different actions may produce differing results – opening the book may cause something to fall out, while reading the volume may enlighten you as to the written contents.

Mirrors should be looked in, moved, looked behind, rubbed, manipulated so that they shine at something, and, if all else fails, smashed. A hammer is a handy breaking tool and it might also be useful for banging in or removing, or driving stakes through the hearts of vampires.

One adventure used such an implement for getting at what lay beneath a nailed-down carpet.

Flies in the ointment

CERTAIN objects have a limited life, so you may have to try and make use of them quickly.

For instance, a lit candle may have to be brought into service before it burns down, especially if you had no matches or other means of relighting it if it was blown out. Icicles and other frozen matter may not last long in a warm environment.

Some objects such as fragile vases do not take kindly to being dropped on a hard surface so endeavour to let them

Utilising objects

down gently – a cushion may be the answer. Some artifacts may have less than obvious uses, while others may require additional items to be added to them in order to construct a larger and quite different item.

In one adventure, a hammer, some wood, nails and canvas were all required to make a ship. In another, an empty wine bladder and some swamp gas were needed to make a bomb. In the recent Guild of Thieves, a billiard cue, some thread, a needle and a maggot were all prerequisites to a DIY fishing rod complete with bait.

Cushions were mentioned earlier as a possible device for protecting fragile items. If the cushion happens to be of the inflatable kind, it could also be used to store liquids or gas, act as a springboard or even as a lifebelt.

Rope is nearly always useful, but it may not come in the form you expect. So look out for sheets, cummerbunds, scarves, cords, cables, sashes, washing lines, leashes, reins or anything else that can be tied together to make a functional rope. Knight Orc has you tying many such utility items together to form a very long rope needed to cross an abyss.

You have to use your imagination when considering the role of a specific object, as the item's raison d'etre in the game may be quite subtle, yet provided the author has abided by the code of adventures, always fair and logical.

Reasonably large-sized fragments of glass may serve as cutting tools, magnifying lenses or for focusing the sun's rays to start a fire. Other than as light sources, candles could also be useful as a waterproof coating – hot wax allowed to drip and cool – or for making a surface slippery.

Newspapers make good firelighters and possible containers as paper cones. They can really come into their own in that old trick of retrieving a key from the keyhole on the far side of a locked door by sliding the newspaper under the door, pushing the key out with something like a paperclip, and withdrawing the paper when the key falls on to it

Other ways of unlocking doors when there is no key to be had could be by picking the lock with a brooch hasp, paperclip or credit card.

Lateral thinking

APART from dragons, the most commonly encountered creatures in an adventure must surely be mice. Where there's a mouse involved, you're almost certainly going to find an elephant not far away. But elephants are not the only ones scared of mice – you may also find an obstructive person who will run off at the sight of a rodent.

There could be other uses – I once used a dead mouse to bung up a hole in a canoe. Where there's mice, you are also likely to find cheese, which itself may be needed to feed another hungry animal.

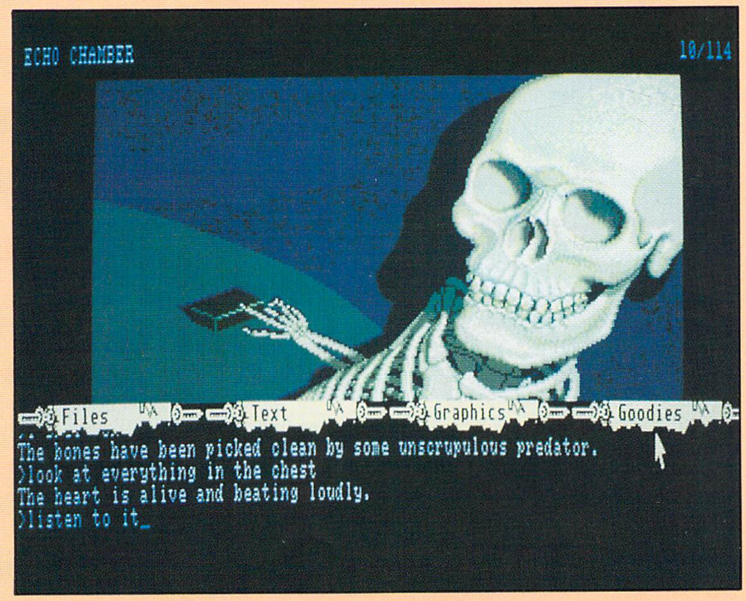
You might think that mud and dirt could not possibly help you in an adventure – but you'd be wrong. Either are suitable for camouflaging your face at night and mud in particular makes a nifty poultice, mosquito-protectant and plugger-upper of leaks. The moral is – think laterally and think imaginatively.

Old bones, for instance, are almost guaranteed to be essential for feeding to hungry, hostile canines, while gloves can not only keep out the cold, but can also be used to aid grasping hot, slippery, poisonous or otherwise painful objects without injury.

Finally, you may come across strange words in an adventure – they might be chalked on walls, etched on rocks or scribbled in books. While you can't take them with you in the same sense as material objects, don't forget them. Famous examples are PLUGH, XYZZY and BUNYON, all magical words that, when spoken aloud, cause you to be transported instantly to another location – very handy for fast travelling and for moving possessions around more easily.

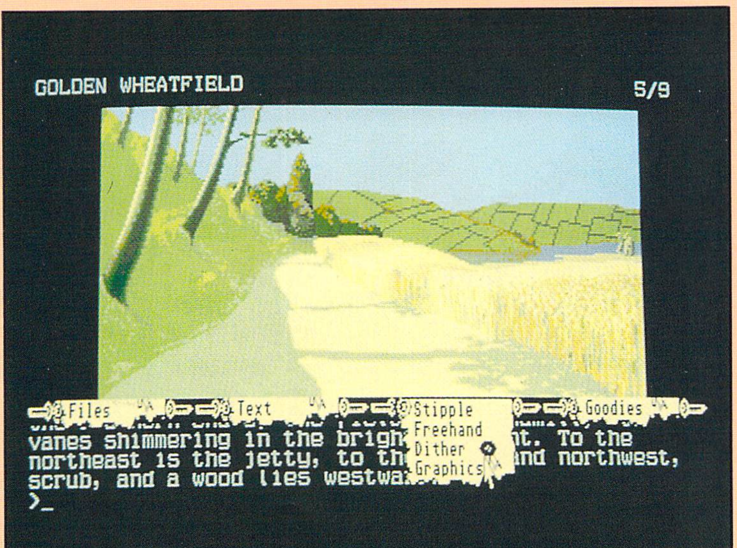
PLOVER was another magic word that immediately transported a particular object, while BLACH caused bells to ring and an important point to be gained when spoken at the right time.

Whether it be a strange word, familiar or unfamiliar object, mark well this rule – take it with you.



```
ECHO CHAMBER                                    10/114
The bones have been picked clean by some unscrupulous predator.
>look at everything in the chest
The heart is alive and beating loudly.
>listen to it_
```

Old bones may provide nourishment for a wild beast



```
GOLDEN WHEATFIELD                              5/9
vanes shimmering in the brigh          nt. To the
northeast is the jetty, to th          and northwest,
scrub, and a wood lies westwa
>_
```

Even a wheatfield may hide important artifacts

Making yourself understood

ADVENTURES have made giant strides since they first appeared on home micros. It doesn't seem that long ago when, in addition to pitting their wits against the often inscrutable or devious mind of the programmer, adventure players were also forced to wrestle with the game's primitive parser.

In those days, a parser was likely to accept only two words of command at a time, inevitably a verb followed by a noun. And that wasn't the only constraint. Limitations of computer memory also meant that the adventure's vocabulary had to be drastically curtailed. For example, even if you knew the correct action to a particular situation was to move a vanity mirror, you may have struggled in vain to find the right combination of words to accomplish the action.

This was not because you didn't know what to do but because the program required absolute precision. MOVE GLASS, PUSH MIRROR, TAKE MIRROR, PULL GLASS and many other combinations would have come to no avail if you didn't hit on an exact match with the program, in this case, LIFT VANITY.

Restrictive or helpful

I REMEMBER a particularly wretched adventure where one of the very first puzzles involved a decorated marble pillar. Doing something to the pillar seemed likely to open up a door to a secret passage. It turned out that the only acceptable response to the puzzle was MANIPULATE SYMBOL.

Since the game hadn't even mentioned that there was a symbol on the pillar, let alone hinted in the documentation or description that the word MANIPULATE was going to be required, the odds were heavily against anyone making any progress with that bundle of frustration.

Such programs can still find their way to the market even now, although thankfully they have become increasingly rare. Most adventures feature far more sophisticated parsers and huge vocabularies, the latter allowing you to use a range of alternative words to achieve an action and still meet with success.

Modern day parsers allow quite complex actions such as TAKE ALL BUT THE GREEN BOX, OPEN THE BLUE LOCKET, PUT IT ON THE LEDGE, EAT THE CAKE AND GO NORTH. The use of ALL and IT certainly saves wear and tear on a tired adventurer's fingers, but ironically enough the occasions you need to use such powerful and complex commands turn out to be far more infrequent than you would have thought.

In fact, most adventurers fall back on the old-fashioned verb-noun even when playing a game that permits far greater sophistication of input. Old habits die hard!

Alternatives to text

ALL adventures, whether they be text only, mainly text with added graphics, or heavily graphics orientated, rely to a greater or lesser degree on the written word for output.

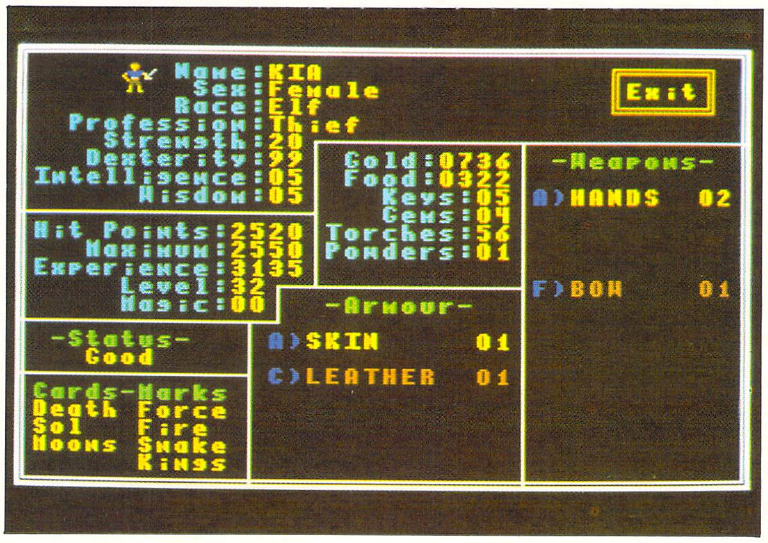
However, to make life easier some graphics-orientated adventures have completely done away with text as input. Instead, they use such things as on-screen command panels, icons or drop-down menus.

Mirrorsoft's Déjà Vu, The Uninvited and Shadowgate are all examples of

A glossary of adventuring terms

THIS is not intended to be a comprehensive list, but should be sufficient to help any novice adventurer get started. It is basically a short glossary of the most commonly used verbs in adventures, together with some examples of any respective and usually acceptable synonyms.

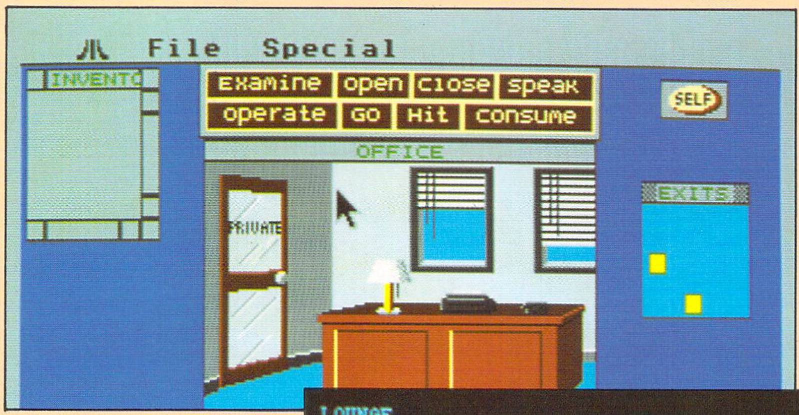
Break	Smash, bend, hit, cut, destroy.
Close	Shut.
Dig	Excavate.
Drink	Sip, taste, consume.
Drop	Remove, leave, put, free, release, lower, empty.
Eat	Consume, taste.
Examine	Search, look, inspect.
Fill	
Fire	Shoot.
Get	Take, grab.
Give	Offer.
Go	Move, run, walk, climb, ascend, descend, follow, swim, dive, enter, leave, fly, mount.
Hit	Attack, kill.
Kiss	
Knock	Tap.
Light	Burn.
Make	Build.
Open	Unlock.
Point	Aim.
Push	Pull, slide, move, tug, lift, raise.
Read	Study.
Remove	
Show	Display.
Sit	
Stand	
Talk	Ask, tell.
Turn	Operate, start, stop, press, switch
Wait	
Wear	



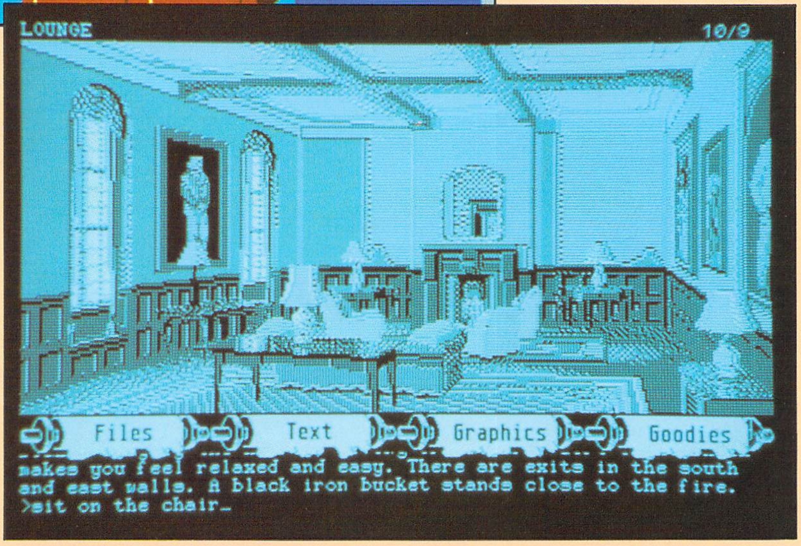
```
        Name:KIA                          Exit
         Sex:Female
        Race:Elf
  Profession:Thief
    Strength:20
   Dexterity:99     Gold:0736      -Weapons-
Intelligence:05     Food:0322
      Wisdom:05     Keys:05      A)HANDS    02
                    Gems:04
Hit Points:2520  Torches:56
   Maximum:2550  Powders:01
Experience:3135
     Level:32                    F)BOW      01
     Magic:00     -Armour-
 -Status-
   Good         A)SKIN      01

Cards-Marks     C)LEATHER   01
Death  Force
Sol    Fire
Moons  Snake
       Kings
```

Status page from a text-only adventure

Making yourself understood



An icon-driven adventure scenario



```
makes you feel relaxed and easy. There are exits in the south
and east walls. A black iron bucket stands close to the fire.
>sit on the chair_
```

A typical screen from a Magnetic Scrolls text/graphic adventure

adventures that favour the on-screen panel. In these, a simple set of commands is permanently in view – all you have to do is point at one and again at an appropriate part of the picture to carry it out.

Of course, this drastically cuts down the variety of control you can exercise but the compensation is that you never get hung up or frustrated by the inability to find the right words for a particular action.

Mortville Manor, originating from France, has a similar method. Here all the words you need are provided in lists by accessing drop down menus.

Communication with other characters, a central feature in this particular adventure, is accomplished in the same way.

One list provides the names of people with whom you may speak, another details all the topics on which you may engage the selected person.

A compromise between the keyboard and pointer systems occurs in Rainbird's Legend of the Sword. Here you are given the best of all worlds. Directional movement is achieved by pointing at a compass while character's names and many of the usual words needed in an adventure are provided in pop-down menus.

However, you can opt to enter commands via the keyboard in the normal way – indeed, since not all the required vocabulary is listed in the menus recourse to the keyboard will be essential from time to time.

Adventurers are generally divided into two camps over the method of input.

On one side, those who prefer keyboard commands maintain that half the fun is in the experimentation and finding the right words that the program will respond to and that this method allows greater flexibility and a far more detailed interaction between you and the program.

The other camp maintains that keyboard entry slows down progress and is often frustrating – menus, icons and other devices speed up play and more than compensates for the reduction in versatility and depth.

Whichever side you take, it is encouraging to note that there continues to be a wide range of adventures available catering for all tastes in this area. So whether you're a keyboard enthusiast or a pointer fan, more than one adventure will be right up your street.

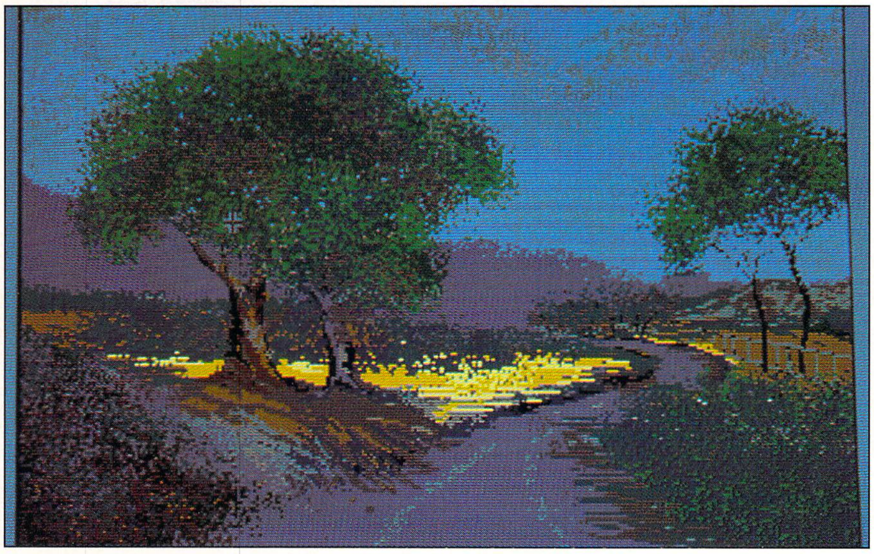
Additional commands

Most adventures have a special set of additional commands for carrying out tasks not necessarily specific to the plot but useful, often essential, to increased enjoyment in playing. Examples are:

Again	Repeats the last command entered.
Brief	Describes location fully on first visit, briefly thereafter.
Help	May provide a clue to a puzzle.
Inventory	Used to provide a list of current possessions.
Quit	Ends the adventure.
Ramload	Loads in a previously saved game from memory.
Ramsave	Saves an unfinished game to memory – faster but lost when computer switched off.
Restart	Begins again from the beginning.
Restore	Loads in a previously saved game from disc.
Save	Saves an unfinished game to disc for later continuation.
Score	Tells you how well you are doing.
Undo	Takes back the last command
Verbose	Full details of each location, no matter how many times visited.

Learning to draw what you see

IF you've ever sat and stared at computer-generated pictures, been amazed by their stunning graphics, and wished you, too, could produce quality artwork on your ST, this section is for you. In it we'll be showing some of the tricks and techniques artists – of all kinds, not just computer ones – use when they create their stunning masterpieces



ANYONE can learn to draw on the ST. It really isn't all that difficult. What is difficult is the discipline required to learn to see what you are looking at.

Learn to see the world around you, how things are constructed and how the whole can be broken down into basic shapes as well as separate parts.

Look at the relationship of the parts and how they form the whole. Although simple shapes may be a good starting point, that is all they are. To progress further you need to look and understand what you see, not just accept the shapes for reality.

You must train your hand to record what your eye sees, which is only a matter of practice. And don't be afraid of making mistakes or be discouraged by them, but learn from them.

Tree drawing

THERE is no secret in portraying trees: It is simply a mixture of drawing and observation skills, combined with a knowledge of colour.

Each species is different. The structure of the trunk plus the growth pattern and style of the leaves varies. To understand the various forms, study deciduous trees in winter when the complete structure is visible.

Evergreens are a little more difficult, but if you take the time to look at the lines of the trunk and branches and how the cluster of leaves form you will gain a better understanding of what the trees really look like.

Painting

WHEN painting a tree the oversimplified method where solid green shapes stand for leaves en mass can often produce a more satisfactory result than the overworked tree in which the leaves are seen and painted separately.

However, a compromise which varies the texture and tone of the painted areas, combining the use of broad washes of colour for the density of foliage in shadow, with a more flexible and crisp handling where small clusters of leaves catch the light, will generally give a more pleasing result.

Learn by doing

STUDY the series of shapes in this section and draw the outlines for yourself. Add the patterns that represent the foliage. Now draw small sketches ranging from one quarter to half screen in size using these same trees to try your hand at composition.

Do your learning work in either monochrome or a very limited colour palette such as various shades of grey or brown to give a sepia effect.

This way you are tackling one problem at a time and have less chance of becoming confused with what you are doing. Concentrate on the drawing technique with some tonal input and leave the problems of colour usage for another time.

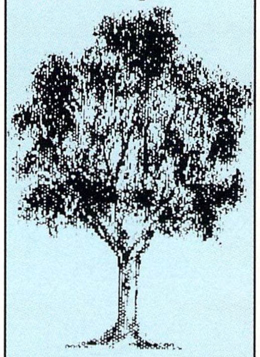
Only when you are content with your ability to effectively portray trees in an aesthetically pleasing landscape composition should you start introducing colour.

Obey the rules

Generally, when people paint trees they draw what they think a trunk looks like and then splash some green around the top area for the leaves, never giving a second thought to the underlying structure of the actual tree they are trying to represent.

1 Watch the shape

TREES don't taper in either the trunk', the branch, or the twig, except – and only – where they fork. Whenever a trunk sends off a branch it diminishes in diameter, and that diameter does not change until it sends off another branch or twig.



2 Above is also below

APPRECIATE that the tree and earth form a whole. Just as the branches of a tree radiate from the trunk in all directions, its roots are groping outwards and downwards into the earth seeking the stability it requires.

You must convincingly convey the impression of the trees being rooted in the soil, not merely placed at certain spots, capable of being moved about like pieces on a chess board.

3 Leaves are different

UNDERSTAND the shape of individual foliage and the patterns in which it grows. Again, each species is different, not only in outline shape, leaves and growth patterns, but also in density of colour.

Learning to draw what you see

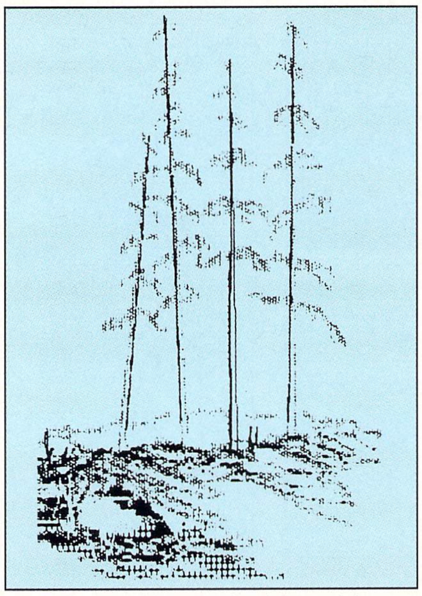
Try this...

WITH continual practice you will achieve the desired effect. Try the following experiments: Draw a single bare tree or outline, as in the accompanying drawings, then block copy this five times. Using monochrome colours and the same tree structure, use different covering techniques to represent the foliage.

Paint the foliage area with varying size brushes. Start again using the spray gun with differing intensities and sizes. Repeat the process, but this time with the stipple option, varying the stipple sizes.

...and this

NOW do the same thing again, but this time use a combination of at least three, preferably five, different tones or colours. Notice how you get the illusion of depth, texture and highlights. Keep experimenting and in this way you will not only learn what doesn't work, but what does.


The trees must occupy the right position...

Position

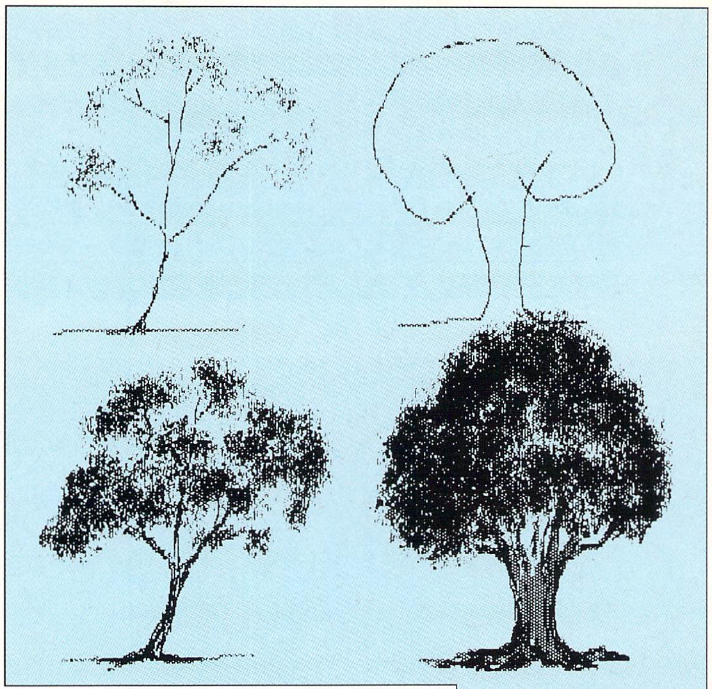
IT is one thing to learn the characteristics of trees and even become proficient at drawing them. It is another to place them correctly into your picture. They must occupy the right position and cover the required area.

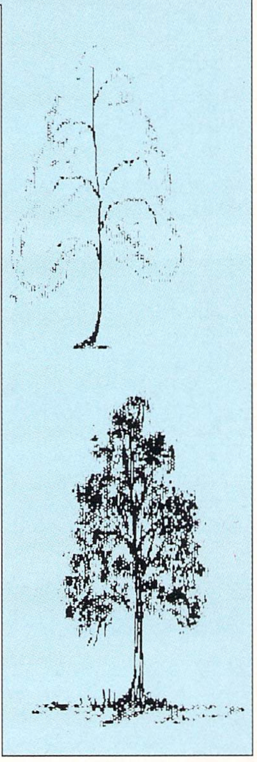
One obvious fault is where strong vertical trees are cut down to fit into horizontal pictures. This tends to disturb the balance, so give some thought to the composition before starting on a picture.

Perspective

IT naturally follows that the larger the number of trees in a group the further away you must be in order to bring the whole within the scope of your vision, and the less detail will be seen in the foliage.

As the distance increases, identification marks will gradually disappear until the whole is resolved into a composition offering the attractions of contour, colour, light and shade, and balance.


Draw the outlines for yourself



Light and shade

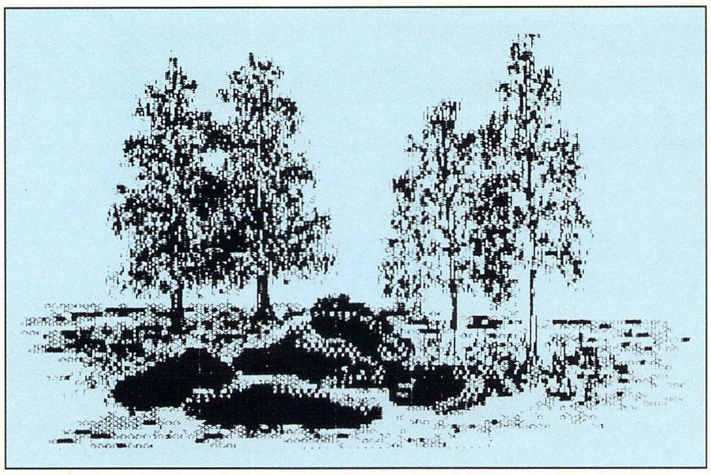
ANOTHER important point is proper lighting to show the position of the sun in relation to the viewer. As a general rule side lighting is preferable to back or front lighting.

Light implies shadows, and these can enrich foregrounds and give depth and mystery to foliage. It plays an important part in directing the eye or giving solidity to your trees.

Size is important

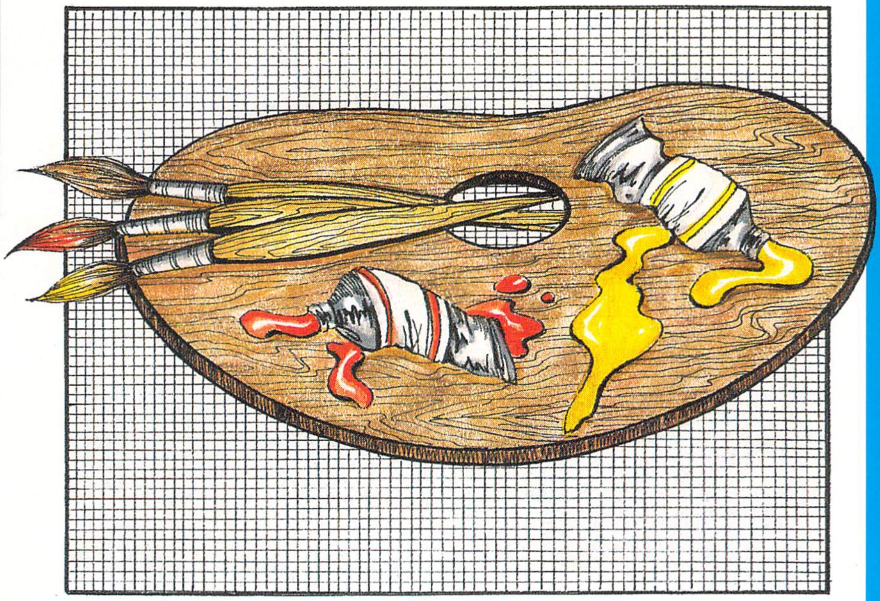
IT is also advisable to give some indication of the size of the trees. This can be done by introducing things such as fences, bushes, hedges, rocks, people or anything that can be easily recognised and provide an instant size comparison.

A composition that consists of trees in a group formation requires to be dealt with greater care than a single tree. Each one must be considered in its relationship to all others as well as to itself, and each may have to be exaggerated or repressed in order to obtain the best effect for the whole.


Composition needs to be dealt with carefully

Freehand drawing and painting

NOW we will paint a simply picture using freehand drawing, flood fills and a small amount of airbrush work. The layout will take approximately 10 minutes. The final stage involving the details within the picture could take anywhere from 30 to 60 minutes, depending on how quickly you work and your eye for colour and detail.



Palette is important

THE most time-consuming task about producing a picture tends to be the creation of the palette. I know some people who may spend a week just setting up their palettes, so don't expect to get yours correct in five minutes.

Each colour you're going to use must be right, otherwise the hues within the picture will not be compatible. You will feel that something is wrong and assume it's the drawing or painting rather than one of the colours.

It's too late to find out after spending hours on your picture that one of the colours is wrong. By adjusting one you can change the colour relationship of your whole picture. The best advice I can give you is to take time to adjust your palette to the correct shades.

Let's get started

Step 1

REMEMBERING the rules of composition, balance, unity and harmony, draw in rough outline your approximate colour areas. Think of this as a rough topographical map – you can see the idea in Figure I (on the next page).

Each major colour will have its own area, and that is what you want to put down now.

As the lines come down the screen they curve, giving the feeling of movement. I did this as it will help to convey the impression of movement at the water line. It will also encircle the seagulls, which is a subtle placement of a secondary interest area, and helps break up what could have been a dead area. The eye is brought to this spot and expects to find something there.

Step 2

FILL in the areas with their predominant colours – sky 677, background trees 055, midground dark sand 554, foreground light sand 776, foreground wet sand 765, foreground wetter sand 554, and the sea 677. Use Figure II as a rough guide.

Now check on the colour balance and harmony. Step back from the screen, squint, and have a good look at what you have produced. If something seems wrong, change it now, as the mistake will not go away, it will only get worse as you continue.

Choosing the colours

YOU need not have all your colours set up before you start, but it is advisable to have either your major colours worked out, or at least half the palette.

The minor colours needed for highlights and darks may be left until they are required.

We will be using 12 palette colours plus white and black – and leave two colour allotments free. Just because 16 colours are available does not necessarily mean that you must use them all.

I could have used fewer than 14 colours, but they were there, and certain lines or highlights looked better in different shades, so I used some of the excess colour space allowed.

The colours I have chosen are very closely related. I had in mind a rather soft picture, therefore I wanted analogous colours. The two predominant ones were green and blue. The greens had to start in the yellow area, and I wanted to have the darkest blue as a blue green. These are all cool colours.

To further soften the colours I have greyed them down to the extent that they form a very neutral or subdued selection. If you look at the colour numbers below you will notice that many of them are in the mid band approaching grey.

RGB value	Colour	Area for use
777	White	Clouds, water highlights and seagulls.
776	Off-white	Cloud highlights and the light sand in the foreground.
765	Cream	The wet sand in the foreground and highlights elsewhere.
444	Grey	The distant water line and touches in the midground bushes.
055	Aqua	Background trees, the water line and bushes.
345	Blue	The water line.
677	Blue	The sky and the sea.
242	Green	The tree tops.
050	Green	The midground.
354	Green	The midground bushes.
264	Green	The touches on the tree trunks plus the mid-ground hill.
550	Green	The touches in the water line and mid-ground.
554	Green	The wetter sand, darker foreground and bushes in the midground.
000	Black	The sea gulls' feet and beaks, touches on the tree trunks.

Freehand drawing and painting

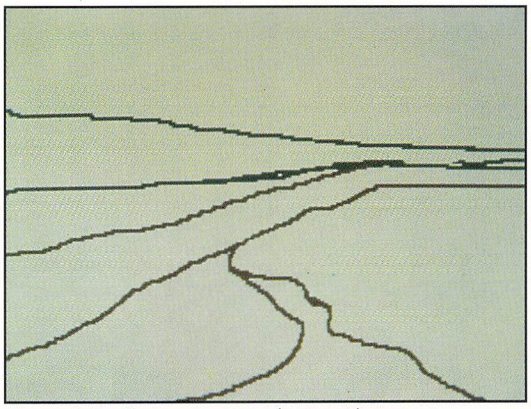


Figure I: Outline your approximate colour areas

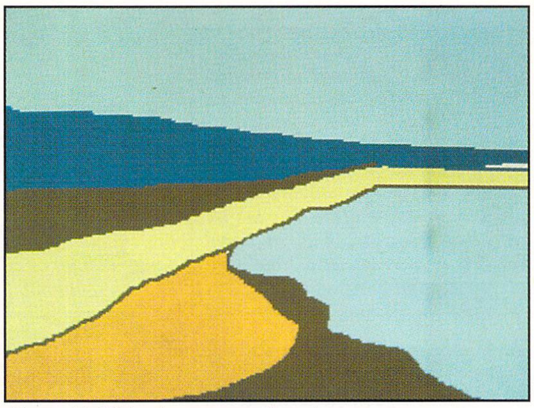


Figure II: Fill in the areas with colour

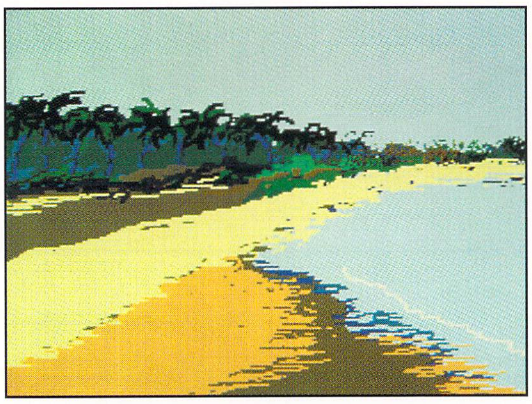


Figure III: Start adding the detail

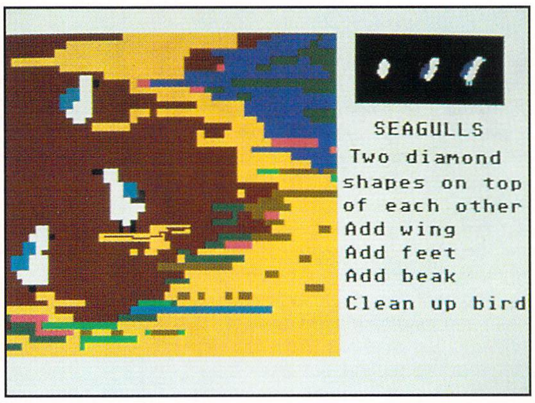


Figure IV: Adding a bit of interest to the foreground

Step 3

FLIP the picture upside down to check your overall balance of line and colour – if it still looks good go on to the next step. If something bothers you about it, work out what it is and adjust those areas that need adjusting.

Sometimes the picture design layout actually looks better when it is flipped. By using your imagination you could come up with a second picture using the flipped version as a starting point.

Step 4

START defining your picture. Using a medium size square cursor draw in the tree tops using three greens (242, 050, 354). Work on the dividing lines to break up the straight lines where two colours meet.

Add some definition here and there, using darker colours – but don't use black – to indicate shadows and lighter colours for the highlights and relief of dead areas. Figure III should give you some idea what you are trying to achieve at this stage.

Step 5

NOW it's time to be particular about what you're doing. Refine your dark and light areas – highlights. Clean up areas that look a bit rough. Break up any colour that looks too solid by adding a spot of another colour to it.

The sky area needs to be broken up. Use the airbrush on a small size and medium flow with white to indicate clouds. Don't overdo it – be light and easy. To highlight the clouds just skim over them with the airbrush using the off-white colour.

The most difficult part of this picture was the waterline. Play around with this using the two blues, aqua, white and the cream colour until you are satisfied with what you have. Use the smallest brush and draw these small straight lines freehand.

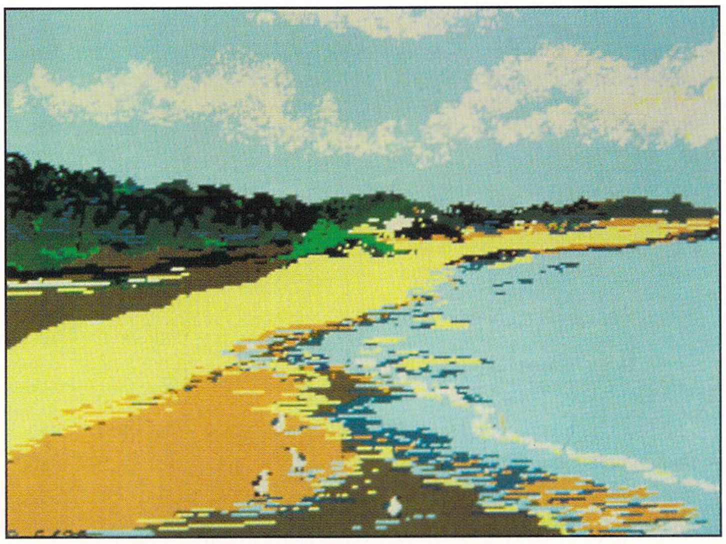
The tree trunks are just a medium thick line drawn on a curve in a medium colour. To give them some solidity I added short strokes of black to bring out certain parts. Don't draw all the trunks in black, as this is too heavy a colour for the picture. What you could do is set up a couple of browns or greys in the two empty palette spots and use them instead.

To add a bit of interest to the picture I have included some seagulls walking in the wet sand by the waterline. Don't be put off – they are really very simple to draw.

Take a large diamond shape cursor and make one point over another in white – see Figure IV: That's the body. Using aqua or blue or grey, add the wing by following one side of the diamond down, starting about three quarters from the top.

Now using black or a dark shade draw two strokes for the legs and one stroke for the beak: That's a sea gull! Experiment with these simple tricks and see how many different poses you can come up with.

Now that you have completed one picture, use the same palette to create another. This time choose a different subject matter and use the colours in a different way.


The finished picture

Creating a palette

The effects of colour

COLOUR is so common in our daily lives that we usually don't give it more than a passing thought. But, for an artist it is of primary importance. Not just the names of the various colours and how to mix them, but how they interact with each other. Their psychological effect on the viewer must also be understood and manipulated to achieve the final required result.

Daylight (white light) contains all the colours of the spectrum: Red, orange, yellow, green, blue, indigo and violet. White is seen because all colours are reflected back, whereas black absorbs all colours and reflects none.

A red object will reflect back only the red rays of light and will absorb the other coloured rays of the visible spectrum. Grey, depending on its value, absorbs a portion and reflects a portion of all white light rays.

Normal pigment mixing theory is quite different from colour mixing on a computer. One is subtractive mixing, the other additive mixing. The additive mixing techniques for finding that special colour you want in your screen picture will be discussed later. Here we will look at colours and how they react with each other so we may better understand how to manipulate them to produce the overall effect we require.

The colour wheel

IN the normal world of art the primary colours are red, blue and yellow. With these three colours, plus black and white, all other colours can be mixed. By mixing red and yellow in equal amounts you get orange. Mixing yellow and blue in equal amounts gives green, and equal amounts of blue and red produces violet. These three new colours are the secondary colours.

Colours such as red-orange, yellow-green, and blue-violet, which are made by mixing the secondary colour with one of its primaries, are known as tertiary colours.

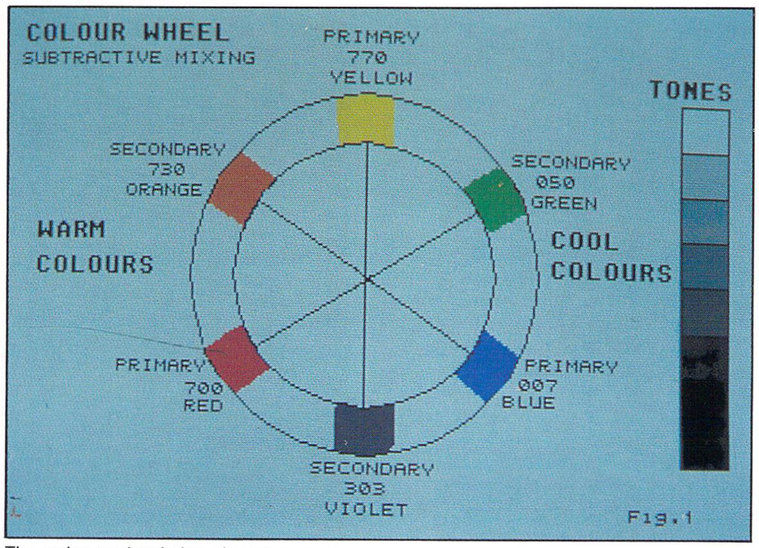
Understanding colour relationships is generally easier if the colour spectrum is set up as a colour wheel. In this, the three primary colours and three secondary colours have been set out. Directly across from each colour is its complement or opposite colour.

Each primary is complemented by a secondary colour, never by another primary colour. Complements are essential in understanding how to use colours.

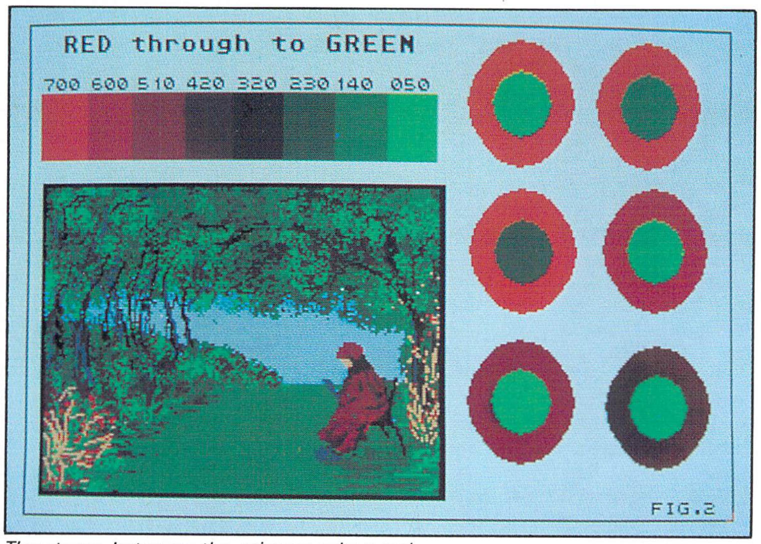
The colour wheel is also divided in half to show the warm and cool colours. The red side of the wheel is considered warm, while the blue side is cool. Yellow can be both a warm or cool colour depending on whether it is tinged with orange (warm) or green (cool). Likewise purple or violet can be warm or cool depending whether it leans towards red or blue.

If you want a colour to look	Put next to it:
More intense	It's complement
Less intense	A more intense version of the same colour, or a near hue
Darker	A lighter value
Lighter	A darker value
Cooler	A warmer colour
Warmer	A cooler colour

How to modify colours



The colour wheel showing the spectrum



The stages between the primary colour red, its complement and secondary colour green

Critical trio

THERE are three principle aspects or dimensions to colour; hue, value and intensity. Hue refers to the specific name of the colour, for instance, red is a hue, so is blue. Value refers to a colour's lightness or darkness (tonal value) and is the most important quality of a colour.

If the value of a colour within your painting is wrong, then the colour is wrong. Every colour has a value, and different colours can have the same values.

Refer to the tonal pictures that were constructed when we talked about composition to see the relationship between tonal value and colour, or look at a black and white photograph that shows something you know to be pure blue and pure red. Both will have the same grey tone.

By referring to the colour wheel you will see that the lightest tones are at the top and the darkest tones are at the bottom of the wheel. Intensity of colour (also known as chroma or saturation) refers to a colour's strength or weakness. The purer the colour the brighter it is.

The same hue can be of maximum intensity (a very bright red) or of minimum intensity (dull red). By mixing a colour with its complement – the colour directly opposite on the colour wheel – we can obtain varying degrees of intensity changes.

The picture above shows the stages between the primary colour red to its complement, the secondary colour green. Both end colours are at maximum intensity and as the intensity

Creating a palette

is brought down by including some of its complement the colour goes through a greying or neutralising effect.

Colour intensity

ADDING complementary colours to each other greys the tonal value of the colour, but these same complements that grey each other when mixed together can be placed side by side or surrounding one another, and they will enhance each other's hue. In other words, complements can intensify each other's intensity.

Strange as it seems, a maximum intensity colour isn't as intense by itself on a white background, as when directly next to or surrounded by its complement. Going further, the same colour directly against or surrounded by an analogous colour (red next to red orange) appears less intense.

Draw the colour circles shown for yourself and watch how a colour changes intensity depending on the colour next to it. Now do the same exercise using analogous colours – those next to each other on the colour wheel. While you change the colours watch how each one interacts with the others.

Some colours in their brightest or most intense state are dark in value – for instance violet, and others light in value like yellow while others are somewhere in between. Place yellow and violet side by side, and they will contrast each other not only by their colour complement but also by their value.

Blue and orange are not as great in value contrast as yellow and violet, while red and green, at full intensity, are about the same value. These last two colours, in equal areas and intensities, and in juxtaposition, can cause some rather strange visual phenomena.

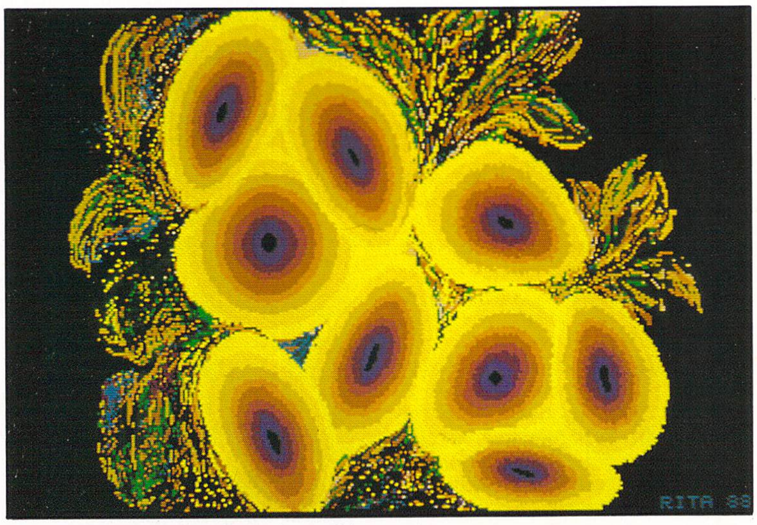
Where the edges meet, the colours seem to oscillate or vibrate. If the area of red is large and the area of green is very small, then the green appears to be very attractive, visually appearing to almost jump right at you. However, if a little red is mixed with the green and a little green mixed with the red, then both colours calm down and seem compatible, even harmonious. Orange to blue and yellow to violet don't react with quite as much agitation but certainly do enhance each other's intensity. This principle was used to set up the series of superimposed rectangles shown here.

In summary, it would be safe to make the following observations about intensifying, influencing or modifying a given colour or value:

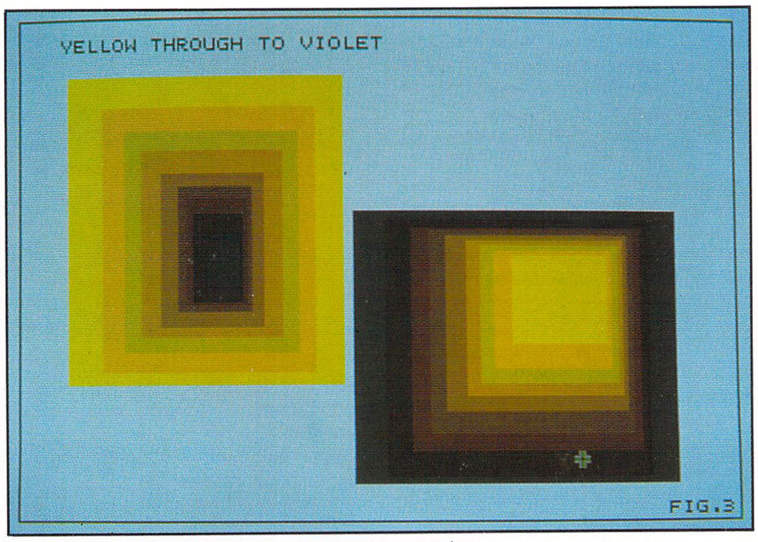
Complements intensify each other when used side by side. Light value colours show up best against dark value colours and vice versa. A greyed colour seems even greyer when a more intense version of the same colour is put next to it. Any colour is influenced by the colour next to it, each tinting the other with its own complement.

The use of colour is a personal and emotional experience. Charts and illustrations are for reference purposes only and should be used as a guide, they should not alter your preferences in individual use of colour.

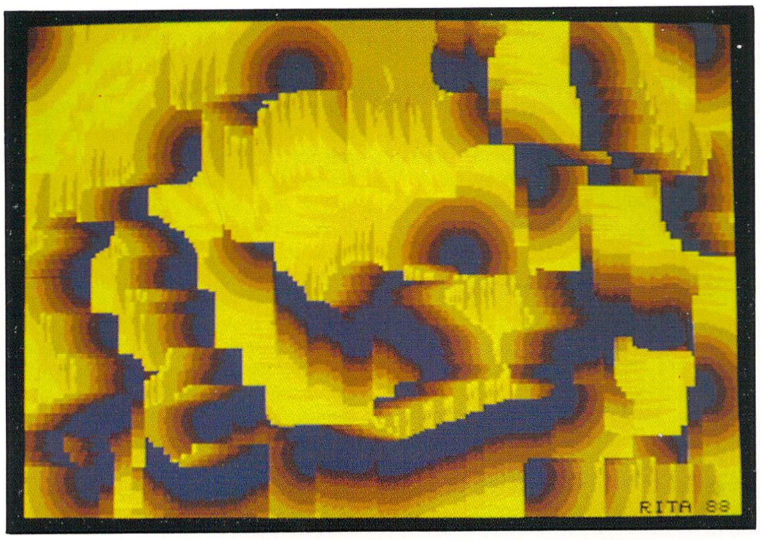
What the charts can do for you is increase your understanding of how various colours work with each other and why, thereby giving you greater control over the use of colour and thus your final picture.



Colour circles showing intensity and interaction



Colour interaction using super-imposed rectangles



Any colour is influenced by the colour next to it

Creating the right colour

WE continue examining the application of colour in computer graphics, with particular reference to its use on the ST. This stage deals with the complicated theme of colour mixing using the ST's palette.

The theory of colour generation

IT may seem strange to anyone used to mixing red and green with a paint brush and obtaining brown, to read that red and green make yellow. But this is exactly what happens when you mix red and green light. Coloured paints and coloured light combine in different ways to produce different colours.

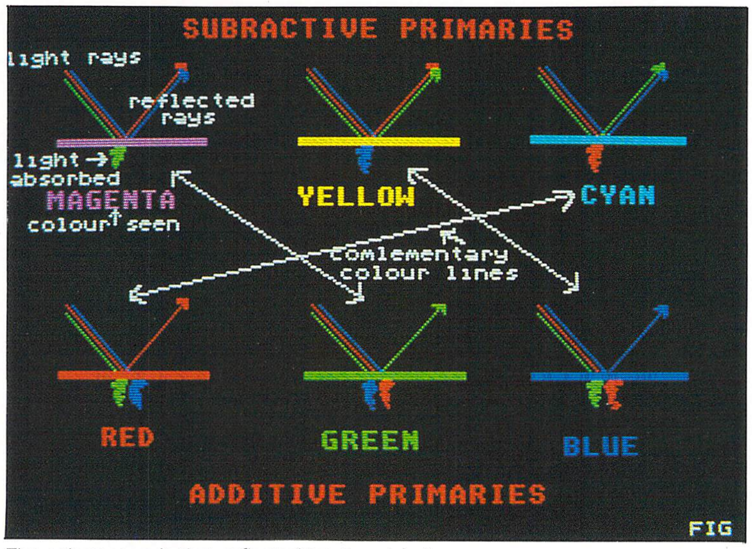
The screen of a colour computer monitor only contains patterns of red, green and blue dots or stripes. At a distance these dots merge into a coloured picture, but if you examine a yellow area of the screen with a strong magnifying glass you will not see yellow, but a tapestry of red and green stripes which combine to give the illusion of yellow.

Try the magnifying glass on different coloured areas to see what colours are used to produce the one you actually perceive with your naked eye.

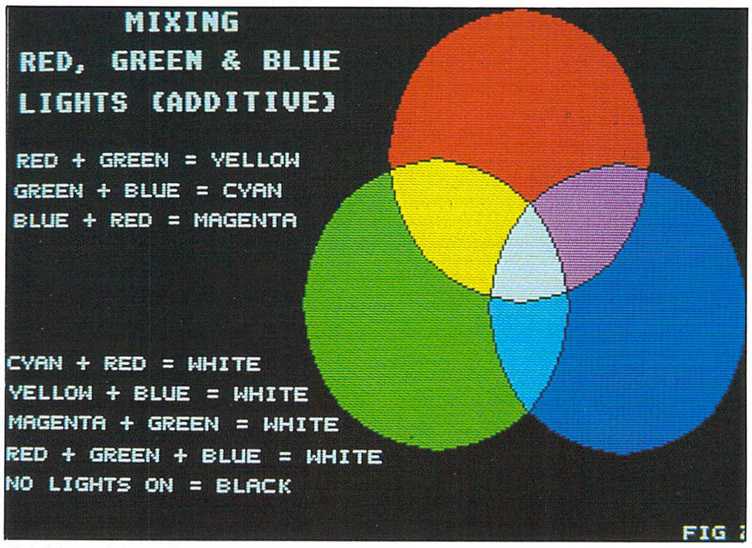
When mixing paints, dyes and inks, subtractive colour mixing occurs. This is what most of us are used to. What we don't realise is how we actually perceive colour.

Colours are generated by the object absorbing some of the light rays from the light source that illuminates the surface of that object, and reflecting others. You see the reflected light.

Thus yellow absorbs blue from the illuminating white light but reflects green and red, which combine to reach the eye as yellow. As shown in Figure I cyan absorbs red light, leaving blue and green to mix and produce cyan.



The colour seen is that reflected by the object



Additive mixing of coloured lights

Mixing colour

So much for the theory of colour and how it is generated. For further information consult an elementary science text book. If you mix the "scientist's" or additive primary colours (red, green and blue) using paint, you will end up with brown, but mix these same colours with light beams and you will get white light, displayed in Figure II.

When an artist wants green he mixes yellow and blue paint. But mixing yellow and blue light will give you a white light.

The theory behind mixing colours with light is really quite simple, all you need to remember is that all coloured areas on a screen are made up of a mixture of three coloured lights. These very small dots intermix to create a third colour, which is determined by the concentration of the three primary colours which are are next to each other in the one pixel.

Our eyes cannot tell these small dots apart, we see only the pixel, and so mix all the coloured lights together to produce a third colour.

For a more detailed look at how coloured dots intermix you could look up the theories of Impressionism, Pointallism or Op Art using dots of pure colours to generate secondary and tertiary colours, at your local library. Red, green and blue light beams mix additively to produce a white light, shown in Figure II.

Where two of the primary colour beams overlap, the additive secondaries of yellow, magenta (bluish red), and cyan (greenish blue) appear. If you mix a primary with a secondary colour and the end result is white you have found a pair of complementary colours, because the complementary of a primary will use the two coloured lights not used in the primary colour.

For instance, the complementary of red will be the result of green plus blue, which is cyan. Where there is no light, the colour produced is black.

To clarify this in your own mind study the tables overleaf and then generate the colours in the tables using your art package.

Adding additive primary colours

ADDING red to green produces yellow, adding red to blue produces magenta, and green plus blue gives cyan. Figure II shows this diagramatically.

To further illustrate practical mixing we shall mix a number of reds. Set red to seven (full light) and green and blue to zero (no light). By leaving the green and blue at zero but decreasing the value of red you will produce various intensities of red until you turn off the red completely (at zero) and produce black (R0, G0, B0, that is no light).

Other shades of red can be obtained

Creating the right colour

by leaving red at seven and adjusting green upwards, giving a variety of red-oranges and red-yellows, until both red and green are at seven to produce yellow.

Then, by having red at seven and green at zero and bringing up the value of blue you go through a series of mauve reds until you reach magenta at red seven and blue seven.

All these red shades can be further adjusted by reducing the amount of the red light, that is, bringing down the value of red from seven anywhere through to one (not zero, you must have some red light turned on if you want to produce a red) and bringing up the values of blue and green.

But, red must be the predominate hue, that is, the value of red must be higher than the value of green or blue. If the green is of a higher value than the red, then the colour will change from red to green.

The only way to really understand this is to turn on your computer and try the exercise for yourself. Watch what happens to the colour with each change of the slide. To alter the colour red you must either increase (add) the value of red to make a lighter tint of red or decrease (subtract) the cyan by decreasing both the blue and green by the same value to give a darker shade of red.

If you wish to keep the red at the same tonal value but change the hue value you have to both increase the red and decrease the green and blue by the same values. This is a very important consideration when taking a hard copy of your screen via a colour printer. Unless you adjust your colours in this way you will find that they will not remain constant.

The red family of colours includes the pinks, oranges, mauves and warm browns. The green family includes the yellow-greens through to the greeny browns. The yellows give various mustards and warm greys. The blues produce mauves and blue greens.

If you have all three slide bars at the same values anywhere along the bars you will produce from white (R7, G7, B7) through six shades of grey to black (R0, G0, B0).

R	G	B		
7	0	0	Red	The three primary colours
0	7	0	Green	
0	0	7	Blue	
7	7	0	Yellow	The three secondary colours
7	0	7	Magenta	
0	7	7	Cyan	
7	7	7	White	All lights on
0	0	0	Black	All lights off

The RGB values of primary and secondary colours

Making a colour cube

A GOOD learning exercise is to produce a colour cube. This would help you to understand how to find the exact colour you require in your picture. Figure III shows one of the six sides needed for the cube.

Square one – the Red Green square in Figure III – uses all the combinations of red and green light. Note that the darker reds and greens tend towards the warm browns.

Square two would be Magenta Cyan which adds a blue value of seven to each square. The other four faces of the cube are: R0-R7, B0-B7 Red Blue. Adding seven green to each of these squares produces Yellow Cyan. G0-G7, B0-B7 Green Blue.

Adding seven red to each square gives Magenta Yellow. To make a real cube you must flip the cube horizontally when you add the colour that is missing, that is, when going R0 to R7. I have produced all combinations in the Red Green square.

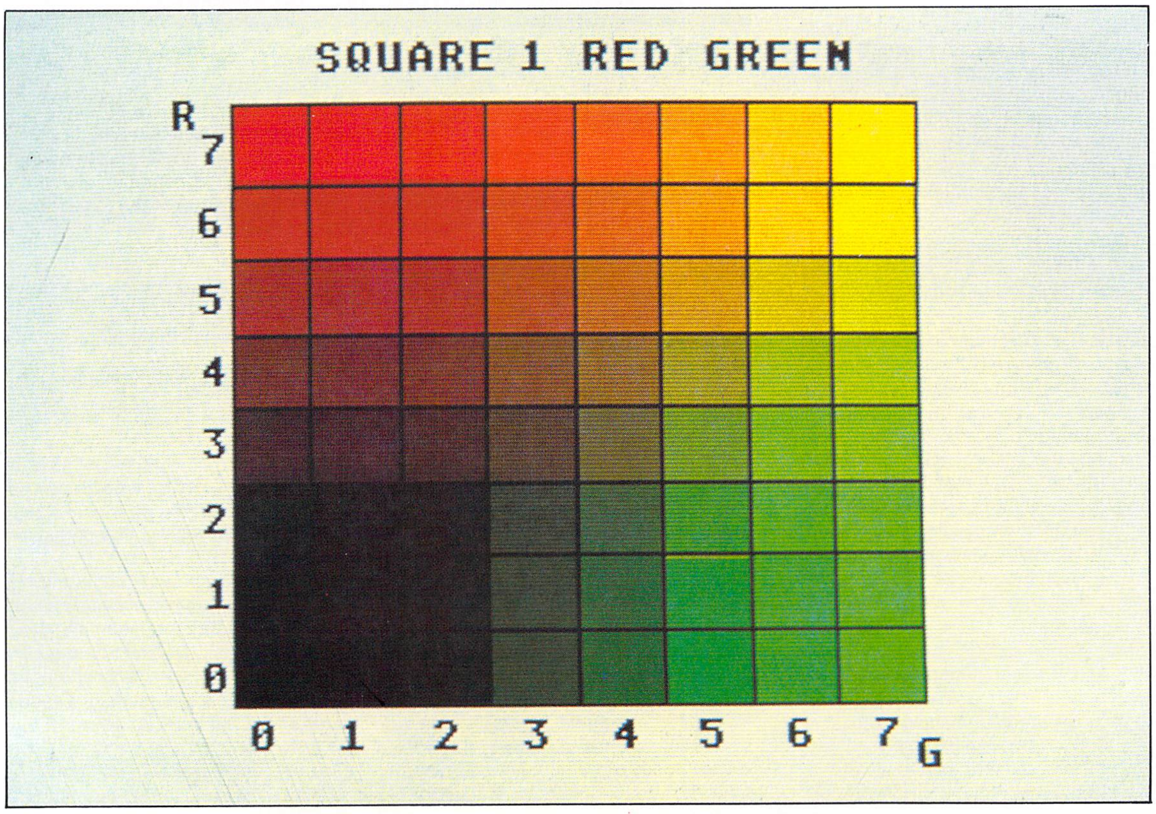
If your art package only generates 16 colours you could set up your square on a four by four grid and use a step of two in the higher values. Any one working with a colour printer or colour digitiser using RGB filters would find this exercise invaluable.

After having worked through these exercises you should no longer be confused about how to produce any given colour. If ever in doubt either add the complementary or subtract the complementary colour from the general colour you want.

This should give you a fair indication of where to look for your chosen colour. It is generally not understood that colour mixing, in any medium, is an intellectual activity. It requires a great deal of thought and experience to mix colours correctly.

	Red	Green	Blue
Red	Red	Yellow	Magenta
Green	Yellow	Green	Cyan
Blue	Magenta	Cyan	Blue

The effects of mixing primary colours



A colour cube created with Quantum Paint

The structure of a picture

Creating the right mood

WE will move on now to look at a very complex topic – pictorial composition. It is impossible to cover everything, so we'll restrict our discussion to the more relevant theoretical points and follow these up with the practical implications.

Pictorial composition even on a computer screen involves combining the various parts of a picture to produce a harmonious whole. This may sound simple, but is in fact, very difficult to do. Composition is an intangible thing and there are no definite rules, only general principles and practices.

If you study texts on art and painting you will find that they are full of compositional designs. These include the triangle, circle, square and ellipse. There is no need to use these mathematical methods, simply follow the dictates of your own conscious and subconscious.

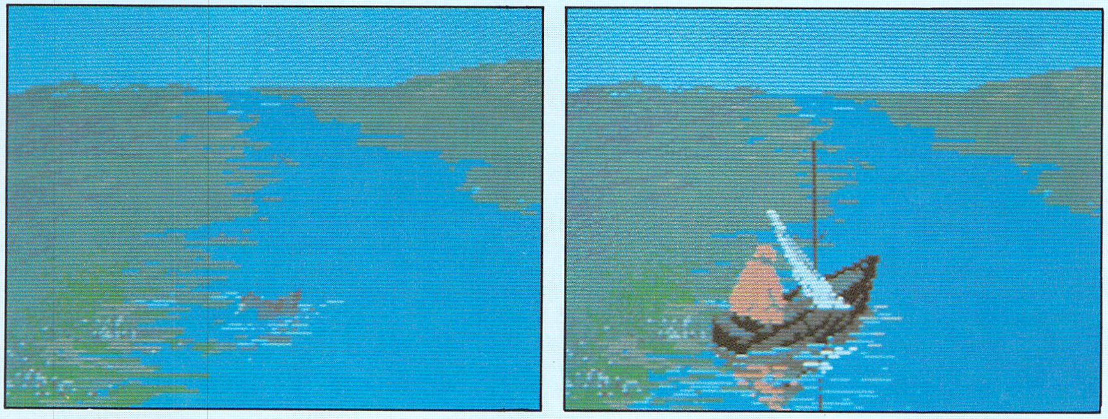
That is, try something, and if it looks right, leave it alone; if it looks wrong, delete it or move it around, or try something else.

This is very easy to do with the ST, especially if you save your work before making drastic changes. Better still, save it every time you produce something pleasing before you continue – and then find that you have ruined a very good effect.

Certain techniques when combined with composition arouse definite feelings, and these can be used to help set the mood of a picture.

But they have to be used in conjunction with the other elements and principles, otherwise the effect will fail and

Ambiguous drawings


Adding an object – the centre of attention – can reduce or enlarge the size of a landscape

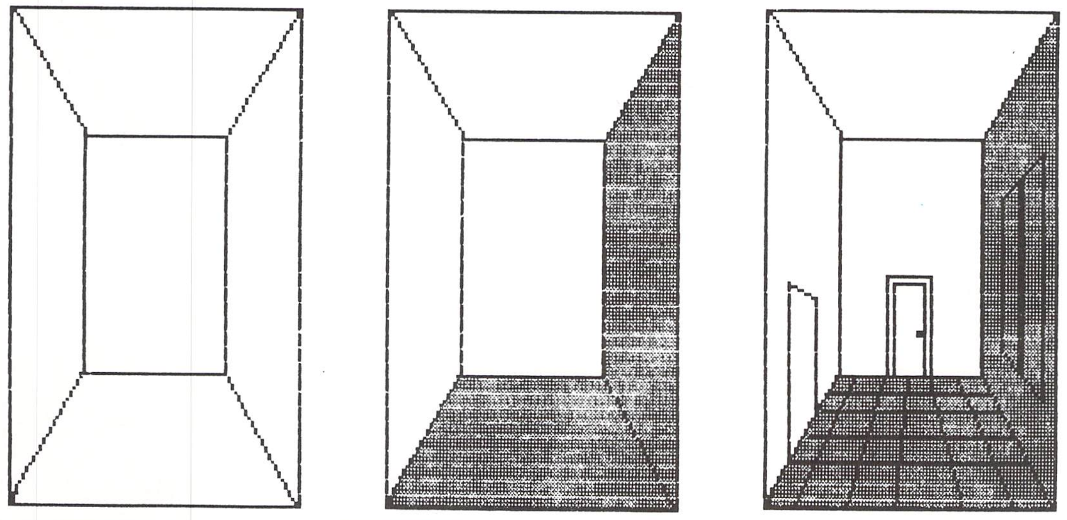
LINE drawings are capable of various interpretations and can be very ambiguous unless we are given some visual clues about them. The figure below shows three drawings. The first is capable of various interpretations. With some extra visual information the second is still ambiguous, but this time it conveys a concave or convex remainder of a pyramid. By adding more detailed information to the third we have now created an unambiguous spatial effect of a room.

The pictures above look at another area of ambiguity – the relative size of the components. When we look at something we want to be able to pin down just what we are looking at. If there is not enough information within the picture our minds become confused and we generally end up unconvinced about what we see.

The pictures show the same scene twice, but by the use of recognisable size comparisons – a duck and a boat – the scale of the entire picture has been changed and we feel more comfortable with the landscape.

The scale is set by the inclusion of the sizing figures, not by the landscape. In the duck picture the landscape has been reduced, while in the boat picture it has been enlarged.

The only difference between the two is the point of interest – the duck and the boat.


Adding detail to a line drawing eliminates ambiguity

The structure of a picture

you will end up with a chaotic mess. Long horizontal lines suggest repose and quiet, and may be used to advantage in peaceful landscapes. Violently broken lines or areas of broken strong colour give the feeling of unrest and emotional turmoil.

This is what is so great about Van Gogh's work. In contrast, El Greco used vertically elongated lines and masses to establish aspiring religious motifs.

Curving rhythmic lines, such as in much Eastern art produce a feeling of gentle movement.

These are just a few of the techniques used to evoke feelings about a picture. Once you have decided on the mood you require for your picture try to capture it on your STs screen.

There are two extremes in the theory of structural composition – isolating or detail composition and central composition. There are, however, many intermediate stages and today any of these midpoints are acceptable.

Central and symmetrical composition underlines ceremonial, sacred themes. These are very strongly based on precise mathematically formulated compositions using the circle, square and so on, and this method of laborious picture construction was used by all the great painters in their day.

Isolating the subject

A PASSIVE type of composition is produced when the subject is isolated and you tend to wonder what is happening beyond the frame of the picture. By giving only part of the picture on the screen and leaving the rest up to the imagination, you are invited to become involved with it.

Rembrandt was a master at this technique, and his subjects were precisely placed in an empty space. He used colour for serenity or mood, and composition for mystery, leaving you wondering what the subject was involved with.

The classical painters of the East were also great masters of the art of using empty space. They actually left blank areas, and at times these have more significance than the painted or drawn areas. The tension and expectation that they created with just a few lines have never been reproduced in Western art.

Another excellent example is da Vinci's Mona Lisa. Ever wondered what she is smiling at? It's not just the mystery of the smile itself, but what she is looking at that gives the picture its power. Some of this mystery was achieved by giving the corners of the mouth the feeling of movement. This was done by leaving a vagueness around the corners and allowing your eye to complete it. This technique was to become the basis of Impressionism.

Impressionism

AN entirely new concept of composition arose with Impressionism. Where previously the weight and stress had always been confined within the edges of the picture, now there arose a disharmony between the incident in the picture and its borders.

When looking at these pictures you feel the urge to pull the subject back into position, and since you are unable to do so your interest is excited and held.

Have a close look at some of Monet's work, especially the waterlily or garden series. It is the same sensation as that given by a piece of music ending on a discord. The passive observer is forced to engage himself with the work of art, thereby becoming an important part of the work itself.

Every picture conveys a feeling which is often aroused by the composition itself, and it is the mystery within it that is the universal trick of all pictures that move us.

The problem is how to achieve this feeling in a drawing on our ST using Degas or Neochrome art packages. It is not easy, and in many cases is stumbled on by accident. The artist's real genius lies in recognising when this accident occurs, and in not trying to alter or improve it. The real art seems to be in omission and in implication, rather than making a complete statement.

Good composition structure

JUST as with a painting, randomly scattered lines on your ST screen may be meaningless, but once properly composed can take on a significant meaning.

Areas of tone give feeling and depth. Line, tone and colour as well as texture – created through colour – must be harmoniously combined if a picture is to be a success.

The general principles within a good composition encompass unity, balance, emphasis and subordination and rhythm. Unity means that all parts of the picture relate to each other to form a complete unit. Everything seems to belong where it is, nothing looks superficial or redundant and nothing is lacking.

Balance means that the picture is in equilibrium and that each part is adjusted so that it receives its correct share of attention. Every segment has a certain attractive force which acts upon the eye, and in proportion to its power to attract, it also detracts from every other part.

If the interest is divided among several sections, if certain lines, tones or colours seem to be too insistent, the composition lacks balance. Your ST's colour palette is an ideal tool for experimenting with this imbalance.

Balance is continually changing as you work – as soon as you add something, or change a colour. It is by continually working to achieve balance that a whole picture is produced rather than a series of separate items combined in one picture.

Emphasis and subordination means creating a centre of interest and allowing nothing to detract from it. The eye must be led to this area and there should be no confusion or major distraction to upset this balance.

Rhythm refers to the regular recurrence of similar features, trees, hills, clouds and so on. This is generally pleasing, as related forms tend to be more satisfying than unrelated ones.

All rules, are there to be broken. Most of the leaders of modern art movements from Impressionism onwards set out to challenge the established rules within art. But before you can break rules successfully you must first know and understand them.

Rules were broken only superficially, and if you study their paintings with care you will find that they do in fact have unity, balance, emphasis and rhythm. They actually concentrated on these areas at the expense of the subject. This is most evident in abstract art, which is really one of the hardest of styles to duplicate on a computer screen.

The only way to improve your compositional work is by practise. Do small quick sketches using a medium to large brush to lay out your line and mass areas. Step back from the monitor, half close your eyes and have a good look at the screen. If it looks harmonious then you have a good starting point to work from. But if it looks like a jumbled mess, start again.

Many artists turn the canvas upside down. Any composition faults, especially balance and unity problems, become obvious, because you are looking at line, colour and tone, not the subject matter. You can do the same thing by flipping your screen upside down.

Another important consideration of any picture is how the objects are placed in relation to the dimensions of the screen. There are boring compositions and interesting ones. The latter contain contrasting shapes – large and small, long and short, round and pointed, straight and curved, full and empty, thick and thin and so on.

But don't use them all in the same picture, just one pair. Contrasts help create interest, but they must also suit the dimensions of the total area used.

Tone and composition

When composing a picture some thought should be given to the background, midground and foreground. In a landscape the background is the sky, the midground mountains, hills, trees, and the foreground is the more detailed parts of the picture at the bottom. By planning this before you start drawing on your ST screen you'll have a better chance of producing a pleasing arrangement.

Using perspective

FIGURE I shows two pictures of the same subject matter drawn on the ST using Degas Elite. The first has been divided into three equal proportions for sky, landscape and foreground. The addition of a barn and a tree placed off centre add balance and interest. But because the complete picture area has been divided into nearly equal parts, it tends to form a monotonous design.

The second illustration contains the same pictorial elements as the first, but this time the frame has been divided into unequal parts and this division represents a more interesting arrangement.

Another area of practical composition is that of perspective, the technique of making your drawn or painted objects on screen look as if they logically belong in the space they occupy in the picture. Figure II demonstrates the five basic rules of perspective:

- Things look larger and more detailed when close, smaller and less detailed as they get farther away.
- Close objects are brighter than those in the distance. Distant objects look vague and hazy.
- An object will overlap what is behind it.
- As objects get farther away they appear higher up as well as smaller.
- Parallel lines appear to get closer to each other as they get farther away. They appear to touch the horizon.

Once you understand these simple rules you can use them effectively in your computer artwork, or you may prefer to ignore them all together if they get in the way of what you want to do. Purposely ignoring rules is a different thing to not realising that there are rules that can be broken or ignored.

Abstract art, as well as other modern styles, are quite difficult to do in a convincing manner, because to consciously break rules you must first understand them. Attempts to bluff your way through without understanding the basics generally produces a confusing or chaotic picture with a very immature style.

Tonal values

Another important area that must be considered is tonal values in composition. Using your art package set up eight shades – tones – of grey from white to black, and three linear shapes – square, triangle and circle like in Figure III.

To draw a square use Degas Elite's Frame option. For the triangle use the Line option and draw one side plus half of the base then block copy this and flip it. Line it up with the other half of the shape and you have a symmetrical triangle. For the circle use the Circle option.

Divide your screen into quarters and place each of the shapes into a number of pleasing compositions to produce four drawings. When you are satisfied with them look at the outlines the shapes produce. There should be a pleasing feeling to the overall positive shape and some balance with the negative space – the background.

Check this by filling in the background with your darkest tone, black. Then undo this by pressing the Undo key and fill in the objects with black. If both versions look acceptable then your placement of negative and positive space is good.

If something feels wrong, change the position or size of one of the objects until you are satisfied with the composition, but save your drawings before you do this checking exercise.

Reload your line drawings and

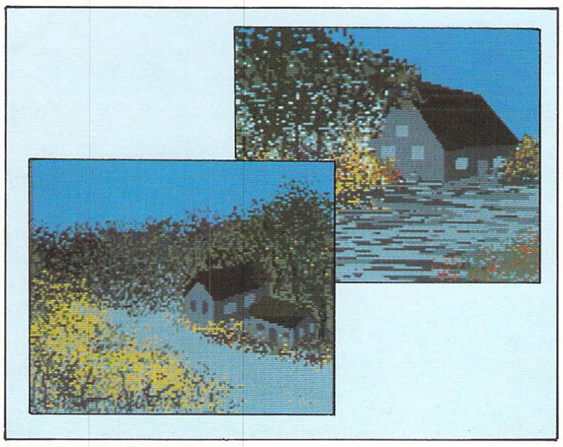


Figure I: Varying balance

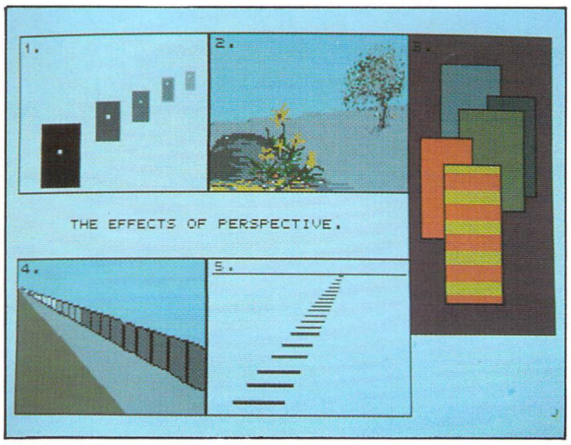


Figure II: Rules of perspective

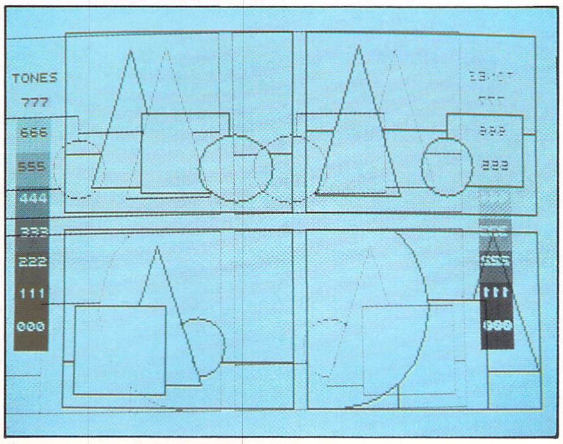


Figure III: Rearranging shapes

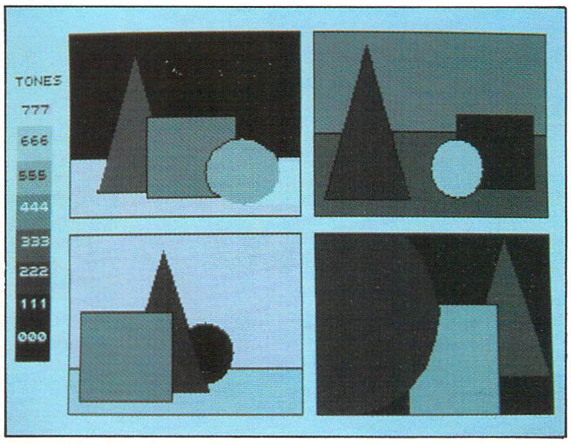


Figure IV: Mixing tone

Tone and composition

colour in each one using a combination of the tonal values indicated at the side of the screen as in Figure IV. Use varying combinations for each picture and notice the difference as you apply the colour.

Now choose one of your tonal pictures and recolour it with different tonal values. As you do this you should become aware of how the picture changes by using different tones. If you have a dark background and light foreground with midtones for the midground, swap the tonal values for both the background and foreground. There should be a marked difference in the feeling of the picture.

Understanding the use of tonal values only comes with practise. As you do these exercises think about the differences in each picture. Experiment further by changing the grey tones to blues, reds or greens. Pay particular attention to how this affects your picture.

When you have achieved a feeling that you like try to match the tonal value to a colour and paint in the various shapes. Do this by block copying your tonal picture and colouring in the copy. Then compare the two. If you have correctly matched tone and colour both pictures should have the same underlying feeling.

Playing with shapes

SO far we have put together some picture designs using rectangles, triangles and circles. Take one of those designs and interpret it in different ways like in Figure V. This is not a standard one you would normally expect when asked to place three objects within a picture.

The circle is represented by the curve at the left border rather than drawing it complete. Althougn you can't see a circle, you still know it to be one. The triangle is standard, but the rectangle, although complete, is placed in a strange position – front centre. By keeping the shapes within the picture constant but by changing where the lines overlap, different and distinct pictures emerge.

The picture breaks some of the rules of good composition by having the rectangle – the centre of interest – at centre front. This is a tricky place to have this – dictated here by the colour of the rectangle, not its shape or placement – but as you look at the picture you will see that this combination does work. Although the composition is structurally wrong, the picture is in balance. This has been achieved by using very strong tonal values to provide the balance.

When you play around with ideas and sketches always try the unusual and unexpected; don't just accept the first interpretation you see in a sketch – be imaginative and daring. If it doesn't work you have lost nothing, but if it does you may have the beginnings of a masterpiece. Be on the lookout for accidents that sometimes occur when you are just playing around, they could be developed into very acceptable pictures.

Most people when they draw or paint, produce the most obvious interpretation, but there are many ways of seeing the same thing. Take an apple. You not only have many different shapes and sizes, but also colours. Then there is the treatment of the colouring itself. Don't just use the Fill option and leave it at that – use shade and texture.

Add a stalk, leaves, blemishes, highlights. Or represent the apple by just the core or peel. How about one with a bite taken out of it, or a grub crawling out? Think of the various things done to apples and when drawing one incorporate some of those ideas.

Shaping up

LOOK for both positive and negative shapes when you plan your picture. They are everywhere you look. Learn to see and recognise them and incorporate them into your design layouts. There are shapes even where you do not recognise shapes existing. Negative ones are some of the most interesting to manipulate. For an effective picture you must be aware of all the space at your disposal and treat each of these spaces with the respect it deserves.

Compare the shapes in Figure VI created using Degas Elite. In the landscape the sky is the negative shape, the land the positive one. Both are as important as each other. Next time you look at one of Turner's later landscapes and marvel at the sky, think about which part of the picture is more important and you will come to understand about the reversal of space.

The second example is an exercise in space. Draw a similar screen and play around with different colour combinations for all parts of the picture, the background as well as the shapes themselves. Recognise how the picture changes depending on which shapes are emphasised by the use of colour.

Try different combinations – start off with tonal variations then become more adventurous and experiment. But, always make your shapes and colours compatible with each other. You do this by first deciding what mood you want to portray with your picture.

By adjusting your shapes, colours and textures to match you'll keep the whole thing in harmony.

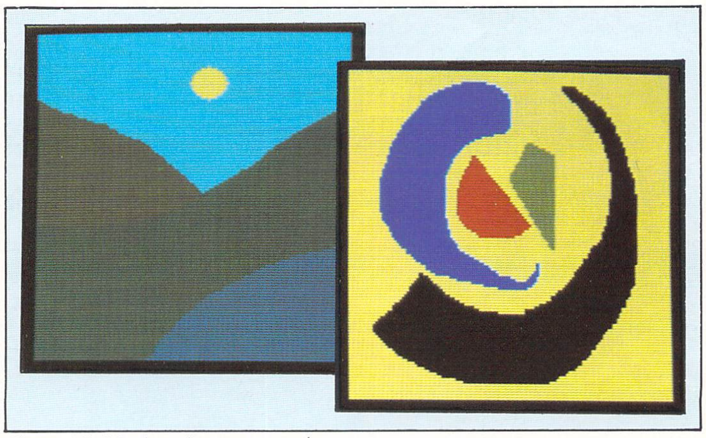


Figure VI: Moving shapes around

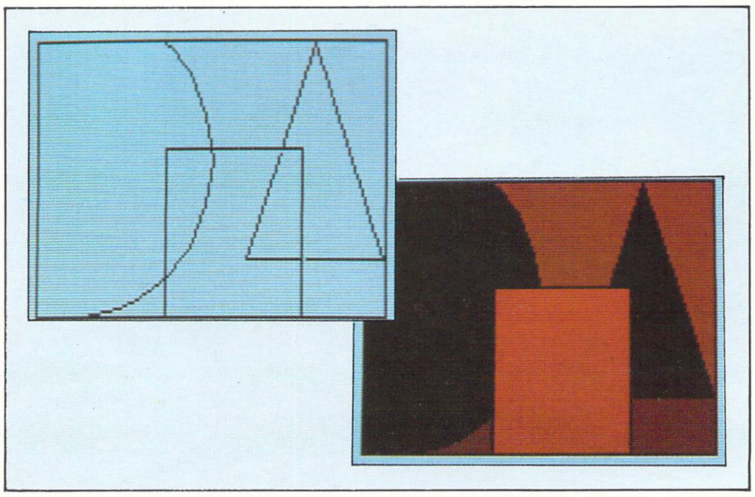


Figure V: Alerting focal points

Creating the right response

IT is well known that certain colours can create specific moods and bring forth emotional responses. This knowledge is used to advantage in designing hospital rooms, industrial work areas and especially in advertising. In the last area colour is used to entice you to buy a product by portraying a mood or symbol that you can identify and associate yourself with.

Psychology of colours

UNDERSTANDING the emotional play of the colours in the ST's wide ranging palette allows you to be in full control of your picture. The emotionality of colour is a fascinating subject.

As a discrete user of colour, the knowledge of how to elicit certain mood responses to your computer artwork can add another dimension, or extra emotional depth, to your picture. It's very easy to create any mood you desire which means you can actually influence the viewer to respond exactly the way you want him to respond.

Red is considered to be the psychologically dominant warm colour, while blue is the dominant cool colour. Red symbolises fire and heat, while blue is the symbol of water, ice, sky and coolness. Therefore the more red a colour contains the warmer the colour will be and the more blue it contains the cooler it will be.

Yellow is considered a warming colour when added to others. It also lightens colours as it is the lightest tone on the wheel we looked at earlier.

When using yellow – a mixture of red and green – as a warming colour it's generally thought of as sunlight, and some yellows on the warm side of the wheel (more red than green), do actually appear to enhance the temperature of the colour to which it is added. A cool yellow (more green than red) will lighten a cool colour rather than warm it.

In creating a landscape on your ST with say, Degas Elite, morning colours are usually cooler than midday and afternoon ones. Late afternoon colours are not only warmer than at any other time of the day, but also can be used to create very dramatic effects.

Early morning colours are cool, crisp and fresh. The sky is more pastel and cooler than the afternoon or evening sky. Each season also has its predominant colours.

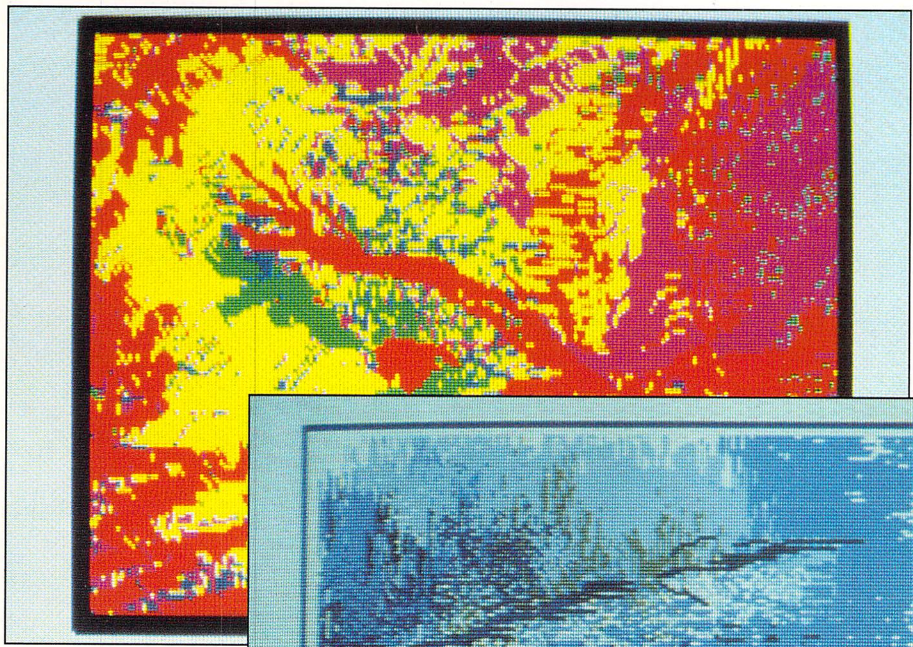
Playing with colour

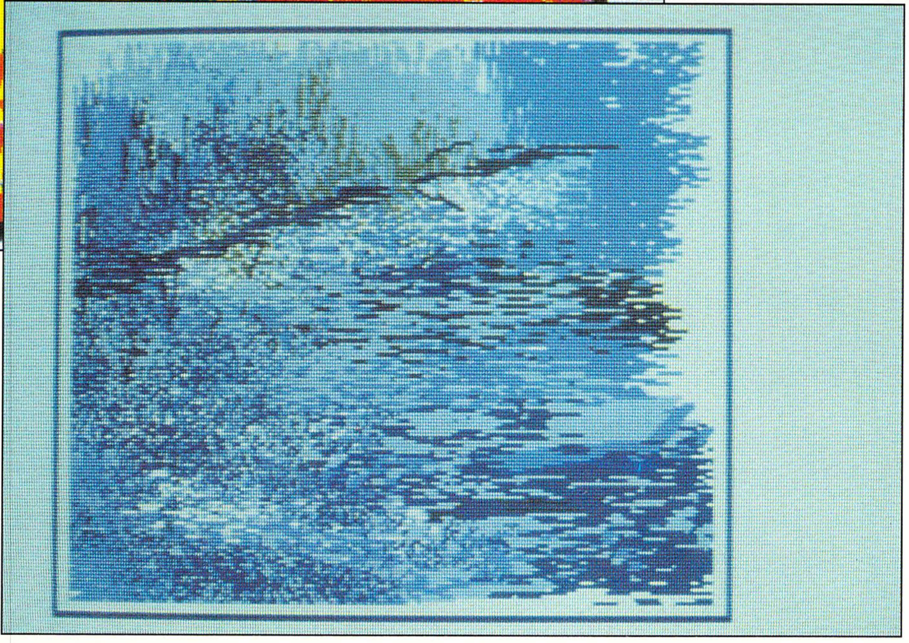
BY painting the same scene, but altering your colour selection by changing the on-screen palette, you can produce quite different pictures. This is done by the emotional play of the colours themselves.

If you have not tried doing this type of colour substitution in your pictures, have a go at it – it's such an easy task on a micro like the ST. You could be very surprised by some of the results, especially when you start incorporating such things as symbolistic and traditional motifs into your work, and combining that with unexpected use of colour.

Expressionistic artwork relies very heavily on colour. Here it plays a dominant role, all other aspects of the picture being subordinate to it. Line, composition, rhythm, realism and so on all play a secondary and minor role to the visual and emotional response to colour itself.

Using the information in the panel as a base on which to build you could portray the following types of feeling by selecting your art package's colour palette accordingly. Take a look at the


Bright, pure complementary colours are loud and unsettling. Try combining these with strong diagonal lines or short, sharp contrasts


Use lightened colours from the cool side of the colour wheel to create a feeling of cool softness. Combine these with thin horizontal lines

Creating the right response

accompanying pictures produced on the ST using Degas Elite, and then try creating your own examples.

When using these guidelines the colours do not have to be pure ones. Yellow itself is a warm sunlit colour, red conveys the feeling of heat, danger or excitement.

The heat of the red can be brought to a climatic crescendo by adding a small amount of yellow – raise the green slider on the palette selector and make sure that there is no blue present. Only a small amount of yellow is required for this – too much and you end up with an orange which does not have the same intensity of either the red or the yellow.

By greying a colour you introduce a quieter mood and, depending on the degree, somberness and discomfort. The aim of the exercise is for you to take a dominant colour and slightly change it by adding another colour to it.

Instead of mixing complementary colours to grey them, try mixing them optically by placing them side by side. Your eye will try to combine them to produce a grey, but because it cannot do so each colour will appear to become brighter and more vibrant.

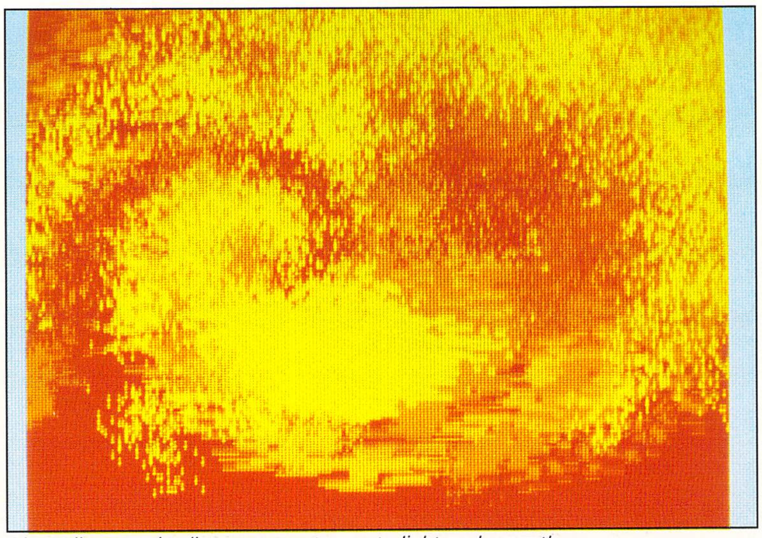
The effect will be even more pronounced as the colours move optically from coloured neutrals – unequal complementaries, one weaker than the other – to grey complementaries equal in intensity.

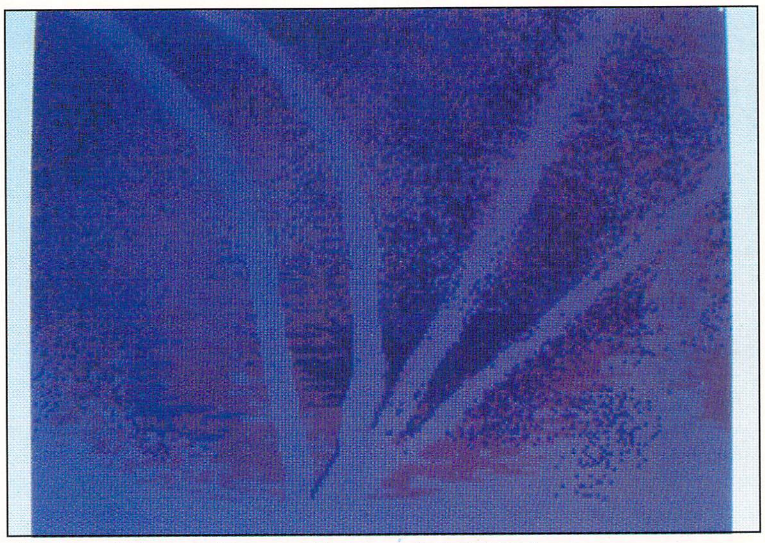
Your eye becomes stimulated as any two colours approach a complementary relationship and this can be used to advantage to produce all types of effects, including very dramatic ones.

For maximum contrast, complementary colours should be in the same temperature scale, therefore use warm colours to warm and cool colours to cool. For instance, a warm blue – blue with some red in it – against a warm orange, or a cool yellow – yellow with a green tinge – opposite a cool pink, violet and so on.

Emotional responses to colour

Yellow	Sunlight, warmth, happiness, comfort
Orange	Light, warmth, happiness, comfort
Red	Fire, heat, excitement, danger
Red/purple	Darkness, intrigue, night, uneasiness
Purple	Darkness, intrigue, night
Blue/purple	Darkness, uneasiness, intrigue, night
Blue	Water, ice, coolness, calmness, moonlight, sky, distance
Blue/green	Water, cool serenity, airiness, distance
Green	Foliage, nature, calming, quietness, coolness
Yellow/green	Sunlight, richness, happiness


Use yellows and yellow oranges to create light and warmth


Purples and blues produce a mystical or mysterious mood

Planning your picture

YOU should first choose your subject matter, then select the mood you want to portray. This in turn will help you to select the correct palette from the ST's 512 available colours. Skilful use of colour will add subtlety, interest and life to your picture and is very often the difference between success and failure.

After choosing your subject, colour and mood, test your palette selection in a series of mini sketches. In each one make minor adjustments to see if you can get a better combination of factors to finally produce the exact result you planned on.

When you've got it right you'll know. There will be no doubt about it, as all the factors within the picture will convey the same meaning. There will be a feeling of completeness to it regardless of whether it is a soft sentimental piece or a strong emotional statement. All will be in harmony.

Last, but not least

A FINAL point that should be mentioned is that the border colour of your picture around the edge of the monitor screen is very important. When your work is viewed it is seen together with the border or background colour. This should be treated as an integral part of your picture.

As you have already seen, colours change depending on what is next to them or surrounds them. Your picture is surrounded by the border colour and as such it has a significant influence on the picture as a whole.

To test this, take one of your pictures in which you have not used the background colour within the picture itself and change the border colour. Look at the effect it has on the picture.

Practical applications

The power of colour

NOW we'll move on to take a brief look at the practical applications of the emotional aspect of colour and composition. To illustrate how you can use colour to say exactly what you want here are two pictures created using Degas Elite on the ST, one a traditional landscape, the other an expressionistic graphical design concept. These should be sufficient to demonstrate the power of colour and how to use it to maximum effect.

Figure I is a green landscape. This is a quiet, sombre type of picture showing a pond at sunset. The main emphasis is on subtle warmth and restfulness. There is nothing here to really grab your attention, except the pond itself. It asks nothing of the viewer and is quite comfortable to look at. This is what is called a rather nondescript picture.

Figure II is identical to the first except for the colours. Here the dominant green has been changed to white and a few of the highlight colours have also been altered to blend in with the overall picture. There is a strong sense of coldness. The sombreness has gone and so has the warmth.

This is a more dramatic scene than the original picture yet the only difference is the dominant colour. The overall colour is stronger, colder and therefore you react differently to the picture. Study both examples until you can work out why you react the way you do.

Try creating your own versions of the two pictures. When you complete the first, save it, then change the dominant colour to give an entirely different picture. You may have to spend some time planning your landscape and how you are going to use your colours before actually painting it, but the planning stages are important and if carried out correctly, usually reward you with a better result.

Feelings within pictures

LOOK at Figures III and IV for a while and work out what they say to you. Investigate your feelings towards them and then try to analyse why you react in that particular way.

The two pictures are drawn in four-colour medium resolution. This restriction can be put to good use as it forces you to think about what you want to say with the picture. Having planned this, decide on the colour combinations.

The difference between the two is quite extraordinary. Not only is the surface changed, but also the whole emotive and symbolic meaning has been altered by changing the palette. That is the power of colour.

The two figures are centrally placed, which gives a strong, forceful composition. The squareness and solidity of the figures has been balanced with the roundness of circles in both corners (the symbolism here is also very interesting).

The strong vertical has been tempered by subtle diagonals to keep the picture within its frame and soften the impact. The placement and shape of the arms, hands and head-dress have also been very carefully thought out. Each part taken separately would look invalid but taken as a whole the picture is not only complete, but makes a very strong statement.

Figure IV is nearly identical to Figure III, apart from changing the colours and the mouth of one of the figures. This small detail helps to change the feeling of the whole picture, and it is more in keeping with the colour changes made.

Purple signifies mystery or a mystical statement. This colour has been used to convey a religious feel to the picture, although the subject matter itself is not one that could be readily identified with our normal preconception of a religious motif.

The red picture conveys a primitive and somewhat threatening feeling, while the same picture in a purple tone – mainly conveyed by the background border – removes the threatening primitive feeling and replaces it with one of strangeness or uneasiness, especially when viewed together with the contradictions within the subject matter itself.

The contradictions within the picture play a major role in the overall effect. Round and square, hard and soft, not

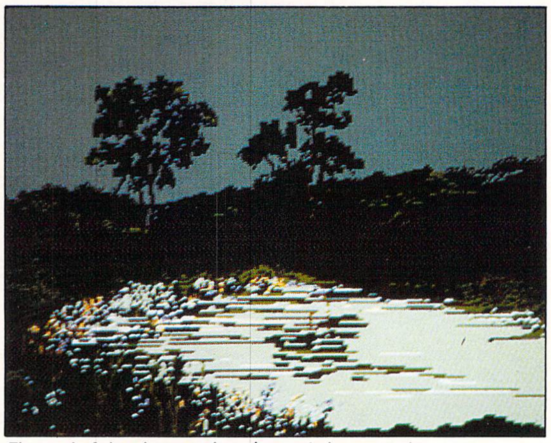


Figure I: A landscape showing subtle warmth

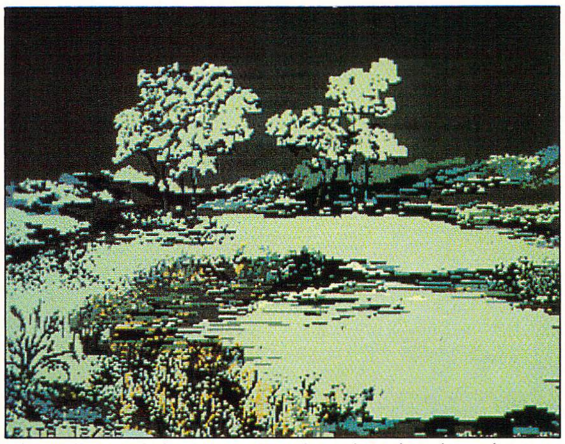


Figure II: Dramatic scene created by changing the palette

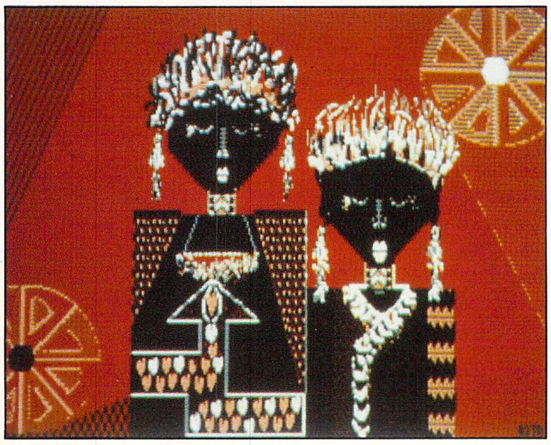


Figure III: A four-colour medium resolution picture

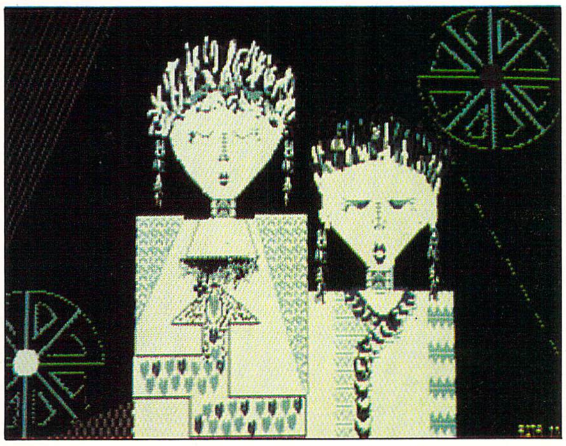


Figure IV: The same picture as before using different colours

Practical applications

only in lines but also colour usage and the subject matter itself. The two pictures show that the number of colours available are nowhere near as important as understanding how to use them to say what you want.

Mastering colour

COLOUR is a servant to but a few. If mastered it becomes an excellent tool. It will do for you what nothing else can. It will communicate directly with the viewer's subconscious and he or she will react to and with your work.

The reaction may not always be pleasant, because you are asking the viewer to do something which he may not want to do, be it think about your work, or even think about himself, which most people generally find uncomfortable or even threatening.

Colour should not be treated lightly. Near enough is not good enough. There is always the one and only right colour for everything. The object should not dictate what colour it should be, but the emotive feeling of the object is of primary importance.

The examples used here should make clear the feelings produced by colour. The topic itself is not an easy one to talk about effectively, the eye and heart (spirit) can understand better than the ear. In other words one picture is usually worth a thousand words.

Tonal composition

HAVING drawn the main picture, think of the best possible composition. Some areas have to be subdued, others highlighted. Use the Degas Smear and Stipple with a small brush to break up any hard background ridges where two colours meet. This softens the colour contours as it merges them.

At this stage some areas of your picture will require highlighting and others will require darkening. Strengthen some of your line work so everything does not merge into the background or sit on the same visual plane. Some strokes can be in the background, others in the midground or foreground.

Where the strokes seem to be placed within your picture will depend on the strength of the line itself. This strength is made up of the colour, thickness and density. By varying these factors you will vary the strength of the line and change the composition of your whole picture.

Now decide on the shape of your final picture. Just because the ST monitor screen is rectangular doesn't mean that your picture has to be the same shape. Try a variety of different things including reversing (flipping) the entire screen. Play around with different types of borders – you'll be amazed what a difference it makes.

Spray it again...

WE will now try some practical exercises in using stippling. Our preference in art packages for the ST is Degas Elite, but it doesn't matter which one you use as long as it has a definable stipple – or spray – option.

Before you start your painting on screen you have to make three decisions:

- Decide on your subject matter.
- Decide on the overall effect or feeling you want to convey.
- Set up your palette accordingly.

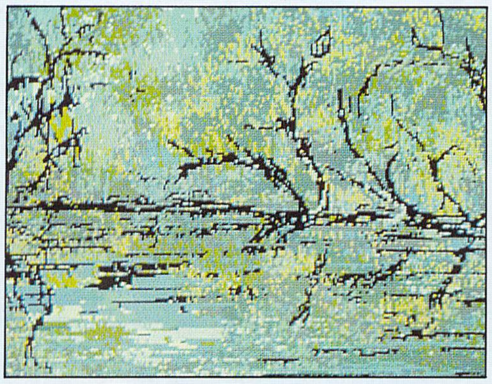
Now select the Stipple option. Using your background colour and a medium brush size, spray around the screen. Allow a heavier concentration in some areas, becoming lighter in others. The stipple density can be regulated by the speed of the mouse. The faster you move it the lighter it is. Move very slowly if you want a heavy flow.

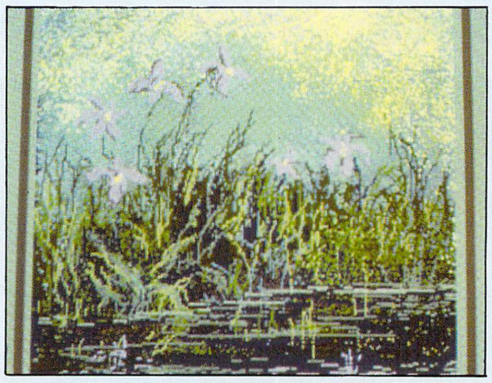
Now change to a slightly lighter colour and smaller brush. Place some accents here and there to give areas of interest and relieve the monotony of overall colour. For variety, try using a long thin vertical brush with the stipple. The horizontal thin line brush gives a wonderful feel when portraying water.

Try a brush at an angle and then go over the same area lightly with the angle reversed. This technique is good for rocks and cliff faces. Try a combination of different brushes – you can get some interesting effects this way.

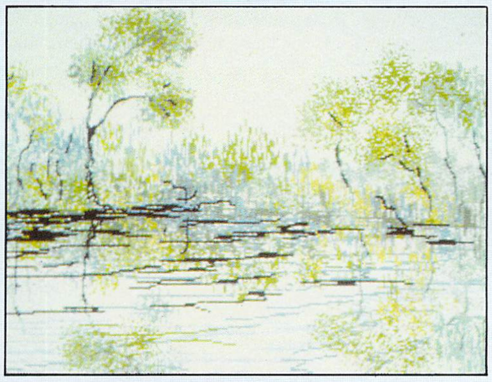
After covering half to three quarters of the screen with a variety of colours and brush lines, change to the Draw option to accentuate areas and bring life to your picture. The size brush you use will depend on what you are doing.

In the pictures below a large brush indicates broad leaves and a medium brush size with a strong colour is used for the flowers. With the trees a small brush with stronger colour contrast delineates the tree trunks thereby bringing them out of the background. How you use colour is a very personal thing, so just try a few different combinations until you have something that feels right.





These three pictures show how to combine the spray and brush to produce effective results



Exploring the 520 STFM

Compiled by André Willey

THE ST range of computers has undergone a number of design modifications and alterations over the two years since it was launched. In this section we take you on a guided tour of the components inside one of the most commonly available variations – the 520STFM

The ST exposed

IF you take a look at the photograph on the next two pages you will see a somewhat stripped down machine, with some of the larger components removed for clarity.

Normally the keyboard obscures the lower half of the main printed circuit board, and the empty space in the bottom right hand corner 1 allows for the mouse and joystick connectors. The internal disc drive sits over the top right section of the board, and the separate power supply board sits neatly over the remaining upper left hand area.

Every ST micro is based around a Motorola 68000 microprocessor 2. This chip – the so-called central processor unit, or CPU for short – is the very heart of the system, and all the other chips on the board are at its command.

On its own the 68000 is incapable of doing anything more than running raw machine code programs, so the ST also contains a full 192k of rom memory, split over six 32k chips 3

Rom and ram

ROM is short for read only memory, and it contains permanently stored machine code routines to allow the ST to provide you with all the things you take for granted like Gem, disc handling, printer support and so on.

Rom memory cannot be altered or re-written in any way, so the only way to update a machine from the older set of chips – shown here – to the new 1.09 variety is to physically remove them and fit a new set. More modern versions of the 520 or 1040 will probably contain just two somewhat larger rom chips.

Similarly, a computer is quite useless without somewhere to store the programs you wish to run. This area of memory is known as random access memory, or ram, and unlike rom memory it doesn't hold any predefined contents.

What goes in here is entirely up to you, and any program – be it a game or a business package – which you decide to load will be stored here until you either erase it – normally with the Quit menu option – or turn the power off.

The 520 and 1040 machines are both based on the same main circuit board, and they each use the same kind of ram chips. Known as 256k drams, each stores 256,000 single bits of information. As eight bits make up one byte, it follows that eight of these 256 kilobit chips will give you 256 kilobytes of storage space.

You will note from the photograph that in fact 16 ram chips are installed in the 520 4, giving a total of two sets of 256k – 512k in all. A 1040, which contains twice as much ram, will have a full set of 32 chips – filling up the currently empty top row of sockets 5

Support chips

LET'S take a look at some of the support chips without which the ST would not be able to function:

6 Glue: The first of the four Atari custom chips, this is basically responsible for keeping everything else running smoothly. It generates the master clock signal of 8Mhz which keeps the 68000 running.

It handles all requests for access to memory and input/output by the various other chips, letting them each have access in turn, and creates the empty video signals which will later be filled in to create your TV picture. In fact, it really can be thought of as "glueing" the whole system together.

7 MMU – Memory management unit: Another Atari custom chip with two main functions. It is used to keep track of up to 4Mb of memory. Whenever another chip – such as the processor, DMA or Shifter – wants to read or write a location in memory, the MMU makes sure the correct address in ram is available to it.

8 DMA – Direct memory access: The third Atari custom chip allows incoming data from floppy or hard discs to be placed straight into memory without passing through the processor.

It must make sure that access to the ram memory is available when the external device requires it, and in some cases – notably the high speed data from a hard disc – it must also temporarily stop the processor from talking to memory to avoid a conflict.

9 Shifter: The final Atari custom chip, this is responsible for creating the multicolour displays the ST is famous for.

The MMU supplies bytes of screen data, along with colour palette information, and the Shifter turns them into RGB – red/green/blue – video signals ready for your monitor. It works in colour only and its operation is muted for monochrome mode.

10 MK 68901 MFP – Multi-function peripheral: Responsible for keeping the 68000 in touch with the outside

Exploring the 520 STFM

world. It provides the system timers, interrupts and control functions.

Among its many tasks are RS232 input and output, screen/processor synchronisation, printer-busy checking and the monochrome monitor detect facility.

11 WD 1772 FDC – Floppy disc controller: A chip responsible for reading and writing data to either the A or B floppy disc drives. It is capable of moving the drive head in or out and then performing the relevant input or output operation – the data being passed via the DMA chip into ram memory.

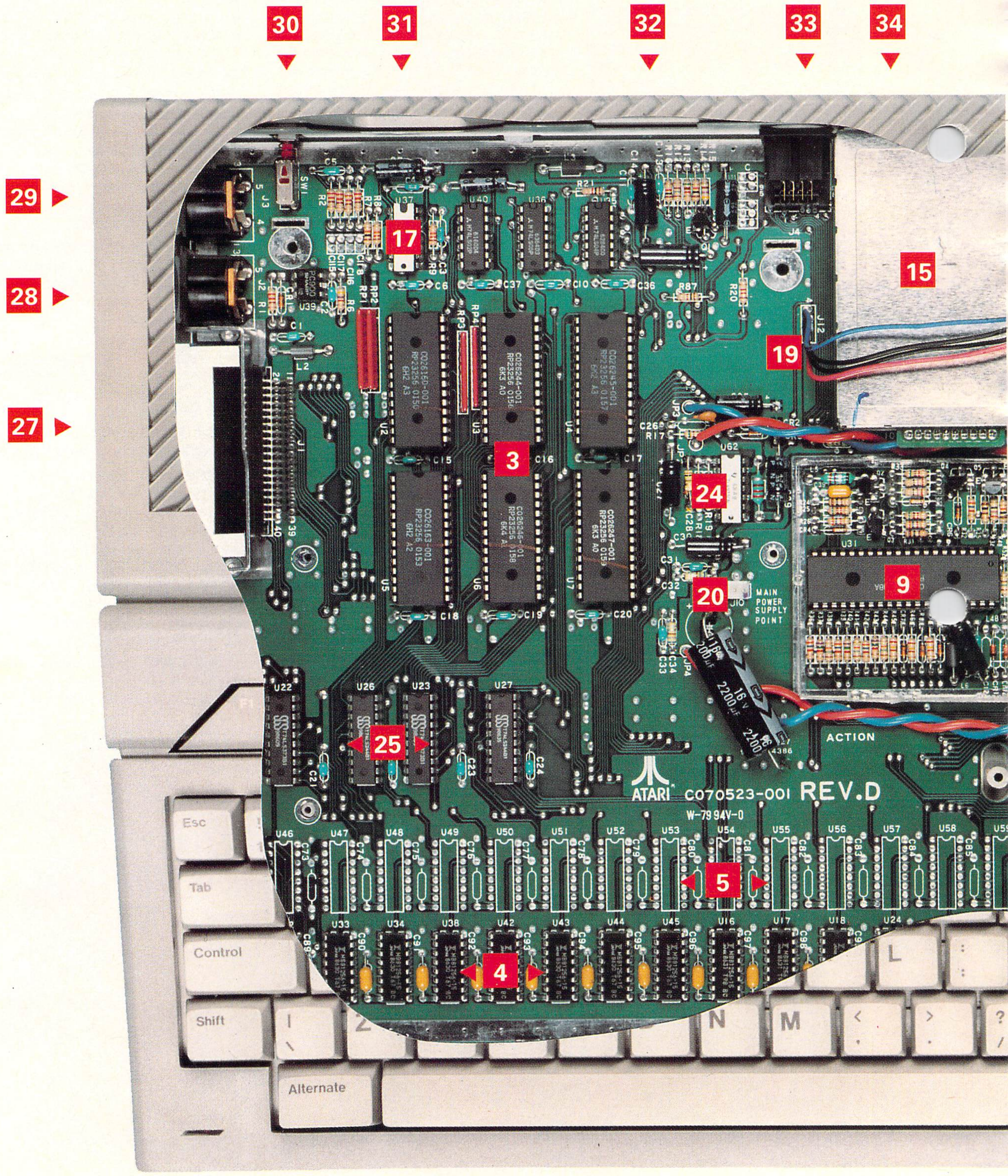
12 and **13 MC 6850 ACIA – Asynchronous communication interface adapter:** Chips used to convert serial data – such as Midi or RS232 – into the parallel type which can be stored in ram or dealt with by the processor.

There are two in the ST: One to receive and send Midi signals, and the other to talk to the keyboard. The keyboard unit contains a processor of its own – an HD 6301 – which handles the keyboard, mouse and joystick and also keeps track of the real time clock. It sends its messages in serial form to the 6850 on the main board for processing.

14 YM 2149 PSG – Yamaha programmable sound generator: Responsible for creating the ST's music and sound effects, and is widely regarded as a rather poor choice for an otherwise very powerful machine. It is now rather old and has been greatly surpassed by a number of other chips.

15 Modulator: Converts the RGB colour signals from the Shifter into an RF – radio frequency – television picture.

It also supplies the composite video signal required for video recorders and some less expensive monitors. The 1040 does not contain a modulator, so



it is incapable of running a composite-type monitor or a video recorder.

16 **Colour/RF circuitry:** Used to create the final video signals.

17 **Reset chip.**

18 **Internal floppy disc drive data connector.**

19 **Internal floppy disc drive power connector.**

20 **Main power supply connector.**

21 **Keyboard connector point.**

22 **Computer master 32MHz clock crystal:** Divided down inside the Glue chip to give the 8MHz signal.

23 **Composite video colour clock crystal:** Used to make colour TV images.

24 **RS232 supply voltage generator (+12v and −12v).**

25 **Ram select latches:** Used by the MMU (25).

26 **RS232 Send/receive data chips.**

27 **Cartridge port.**

28 **Midi-in connector.**

29 **Midi-out connector.**

30 **Reset button.**

31 **Power socket:** For the separate power supply board.

32 **Power on/off switch:** On the separate power supply board.

33 **13-pin RGB/mono/composite monitor socket.**

34 **RF-out TV connector.**

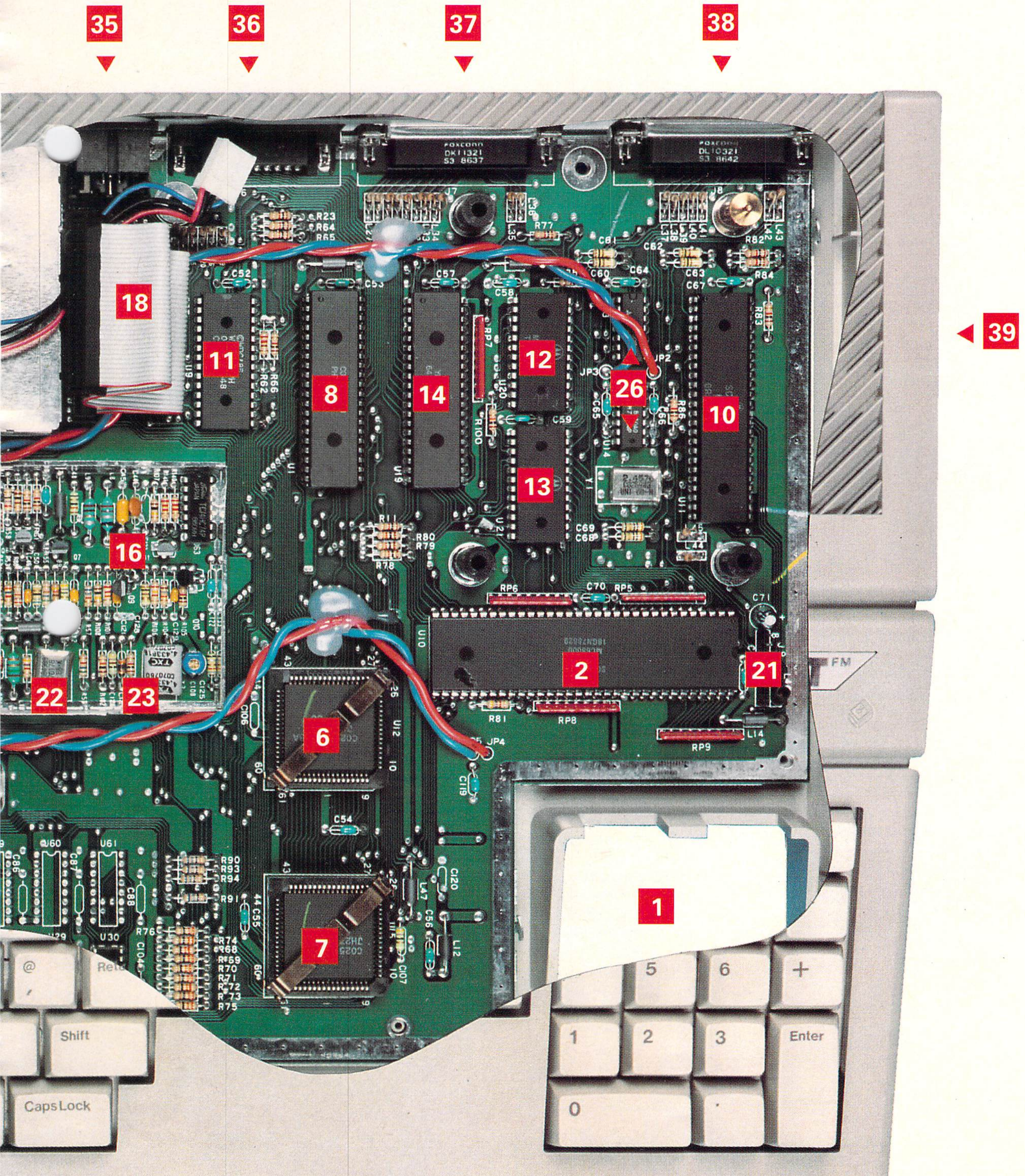
35 **External floppy connector.**

36 **DMA port:** For a hard disc or laser printer.

37 **Printer port:** Bi-directional Centronics.

38 **RS232 port:** For modems.

39 **Internal floppy disc drive:** Sits above the main board.



Exploring the 520 STFM

Glossary of terms

Bit: The single smallest piece of information a computer can handle. It represents either an on or off signal and is normally referred to as having a value of one or zero.

Byte: A group of eight bits of data.

Kilobyte (k): 1,024 bytes – a figure dictated by binary maths. Sometimes data may be stored as a set of bits – in that case 1,024 bits make one kilobit. Not to be confused with a kilobyte which is in fact eight times larger.

Megabyte (Mb): 1,024 kilobytes or 1,048,576 bytes.

Serial: If some data is transmitted in a serial form, each bit of data is sent down a single piece of wire – one after the other – with short gaps between each. This is how the RS232 and Midi systems work, and they only need a couple of wires – one for send, one for receive.

Parallel: Because serial communication is rather slow, a faster method is often used inside the computer itself. This relies on a large number of wires instead of just one, and each single bit within a given byte will be sent down a different wire thus making the data transfer eight times faster.

In fact, the ST is capable of using 16 wires in all, thus allowing the chips to send two bytes of data – 16 bits in total – at the same time. This is why the ST is known as a 16 bit computer.

The Centronics printer connector is an example of 8 bit parallel data since it has eight wires and can thus send one byte at a time.

RS232: A serial data standard used by the ST to talk to a modem or some older printers. The speed of sending data down the wire is defined as its baud rate. For example, 300 baud is 300 bits of data a second.

Centronics: The normal printer connector for the ST, which sends its data in an 8 bit parallel form. Most printers these days support this specification.

CPU: The heart of any computer. This is the chip which does all the work running your programs. The ST uses a Motorola 68000 processor which is capable of dealing with either 16 or 32 bits of incoming parallel data at once – hence ST: Sixteen/thirty-two. Unfortunately a normal ST can only use the 16 bit mode.

Machine code: The fundamental language which your computer understands. It is very simplistic and also very complicated to program. Normally you never need worry about machine code, except to know that Gem and most programs that you will use will be written in it. Even Basic is itself a program written in machine code.

Ram: Random access memory. Used for storing your programs, it contains no data when you turn on the computer, and may be written and erased as often as you like. Whenever you load a program or play a game, ram memory is used to store the machine code. Everything will be lost when you turn the computer off.

Rom: Read only memory. Not alterable, it is permanently set up to contain programs and data which will never need to be changed. Gem is stored in rom memory.

DMA: Direct memory access. A method whereby information can be taken straight from a floppy disc or hard disc, for instance, and placed into memory without having to ask the processor to do any work. Similarly data can be sent from memory to some external device in the same way.

RGB: A video system which allows very high resolution by splitting each colour up into its red, green and blue component and sending the three separately. Used for the Atari colour monitor.

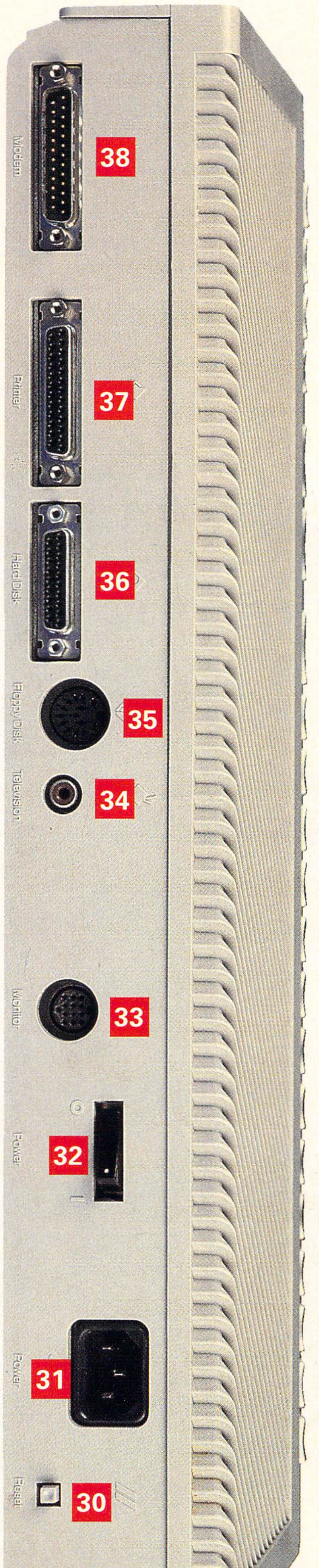
Composite video: The system used by most video recorders to send colour pictures. Also available from any STM or STFM machine.

Monochrome: The very high quality black and white system used on Atari's mono monitors. This runs at too high a frequency to be seen on either RGB or composite video monitors.

RF: The television aerial system uses radio frequency signals, which must be converted from composite video at both the ST and the television end, thus reducing the overall picture quality. This is why medium resolution looks so poor on a television set.

MHz: Megahertz. This is the measurement of frequency often used to judge the speed of a computer. One MHz means one million activities a second, so a clock frequency of 8MHz as used in the ST means eight million clock ticks a second.

Each instruction may take the processor a number of such ticks – or cycles – to complete, so don't think of it as meaning eight million instructions a second.



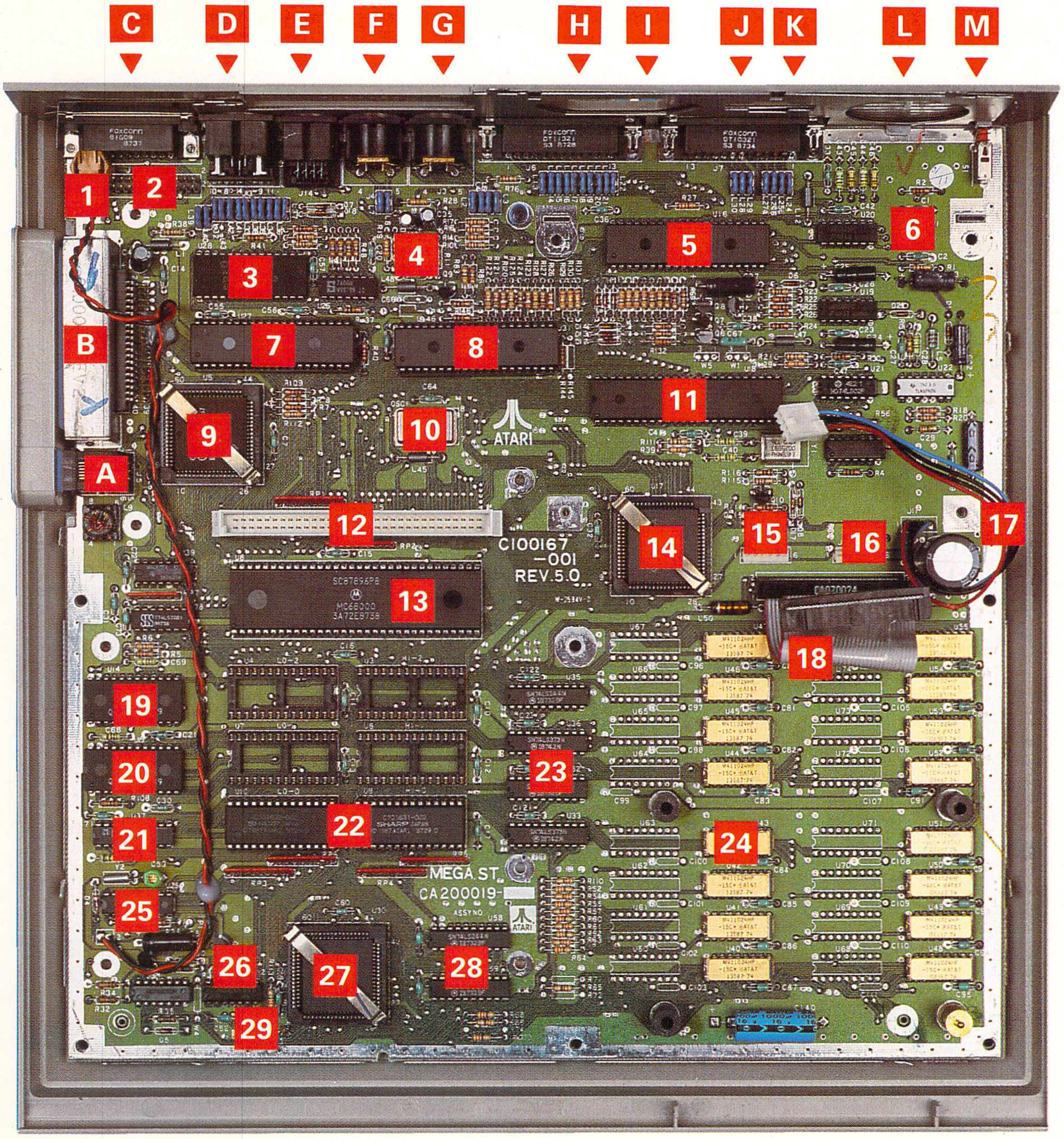
Exploring the Mega ST

Opening the case

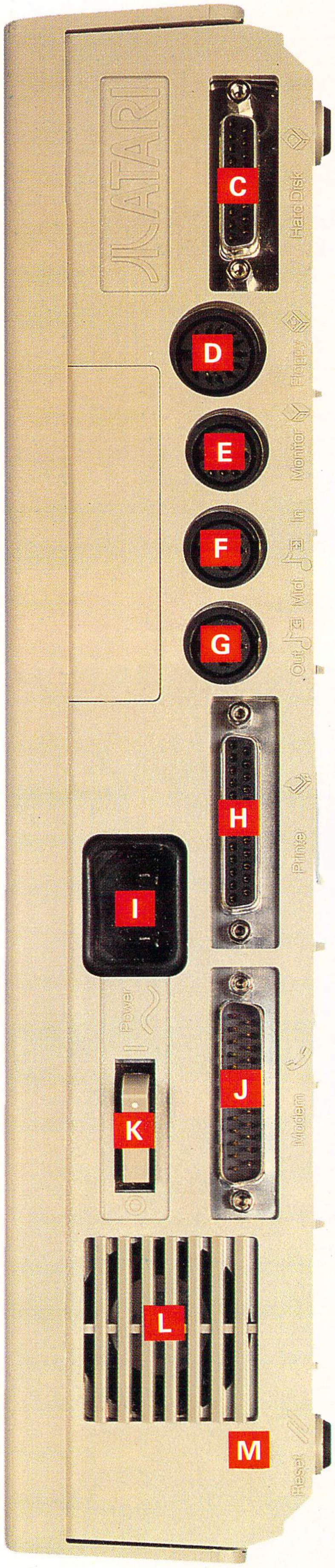
ALL of the Mega-ST range of computers are based on the same main circuit board, as shown in our photograph. Many of the components are the same as those we've already seen in the 520 STFM, so we won't be covering them in so much detail here.

The basic design is similar to the STFM, with the disc drive and power supply board resting on top of the main circuit board. In this case the internal 3.5in drive is located over the lower right quarter of the board while the power supply PCB sits behind it, covering the upper right-hand side of the main board.

The keyboard on a Mega is a detachable unit, giving far more space inside the casing for future expansion. This leaves the whole of the left-hand side of the main board exposed, and an ideal place for third party add-ons to be located after connection to the internal expansion slot.



Exploring the Mega ST



The components

1 Connector for clock battery: This lead plugs into the small battery compartment located in the upper casing of the machine.
2 Internal DMA (direct memory access) connector: This is an extension of the normal hard disc port at the rear of the machine C to allow internal expansion cards such as a hard disc drive to hook straight into the high speed DMA bus.
3 WD1772 floppy disc controller (FDC).
4 PC 900 Opto-isolator: This is a special type of safety buffer which allows data to pass to and fro down the Midi lines without any direct contact with the main circuit board. This is to avoid possible damage to the micro if a wrong cable is attached to one of the ports.

Inside the chip is a tiny LED and a photodiode which work in exactly the same way as the remote control on your television or video. As the light flashes in sequence with the Midi data the photodiode picks up the signal and converts it back to data pulses – all without any physical connection between the input and the output.
5 YM 2149 Programmable Sound Generator (PSG).
6 Reset chip.
7 Direct Memory Access (DMA) controller.
8 Shifter.
9 Blitter: The third of the Atari custom designed chips which, if fitted, allows much faster internal movement of blocks of data. This is most noticeable in terms of graphic speed, and a blitter can often increase the speed of block moves and text printing operations by two to three times. If a blitter is fitted in your machine you will see it as an extra menu item on the main desktop Options menu.
10 32Mhz oscillator: This provides the master clock frequency for the whole machine, which is then fed to the Shifter and Glue chips to be cut down to the more familiar 8Mhz required by the processor.
11 MK 68901 Multi-function peripheral (MFP).
12 Expansion slot: The equivalent of the IBM Expansion Card system, this connector allows you to easily plug in specially designed add-on boards. These could have a variety of functions, but as yet very few are available.
13 Motorola 68000 micro-processor.
14 Glue.
15 Spare power outlet: Connected in parallel with the power supply board 16 this may be used by third-party expansions and other add-ons which require their own power. The supply is rated for 5v DC and 12v DC.
16 Power supply connector: The separate power supply board plugs in here.
17 Internal floppy disc drive power cable.
18 Internal floppy disc drive data transfer cable.
19 MC 6850 asynchronous communication interface adaptor (ACIA): The same as in the 520ST.
20 MC 6850 asynchronous communication interface adaptor (ACIA). This second ACIA is used to convert the serial data from the external keyboard's HD 6301 processor into the parallel form required by the rest of the computer.
21 Real time clock PLA: This converts the signals from the clock chip into a form that the computer can more easily deal with.
22 Rom: The Mega ST normally uses two one-megabit rom chips, giving 256 kilobytes of permanent memory in which to hold Gem and the rest of the operating system. There are six sockets in all to allow for both the older OS roms and for future expansion.
23 74LS244 and 74LS373 ram buffers: These control the data going between the ram chips and the memory management unit.
24 Ram: Rather than using the older 256 kilobit chips found in the 520 and 1040 machines, the Mega ST includes high capacity one megabit ram chips. Eight of these may be used to give a full megabyte of memory – as opposed to 32 of the older type. Thus a Mega 2 will contain 16 of these new ram chips (as shown) and a Mega 4 would have all 32 spaces filled.
25 RP5C15 real time clock: This stores the current time and date used by Gem and the disc handling system. When the ST is switched off the clock is kept running by the battery in the upper casing of the machine.
26 Rom select chip: It handles the various possible configurations of rom chips, depending on which is fitted into your particular machine.
27 Memory management unit (MMU).
28 Ram select chips.
29 Rom selection wire links: They define the size and type of rom chips in use.

External connectors

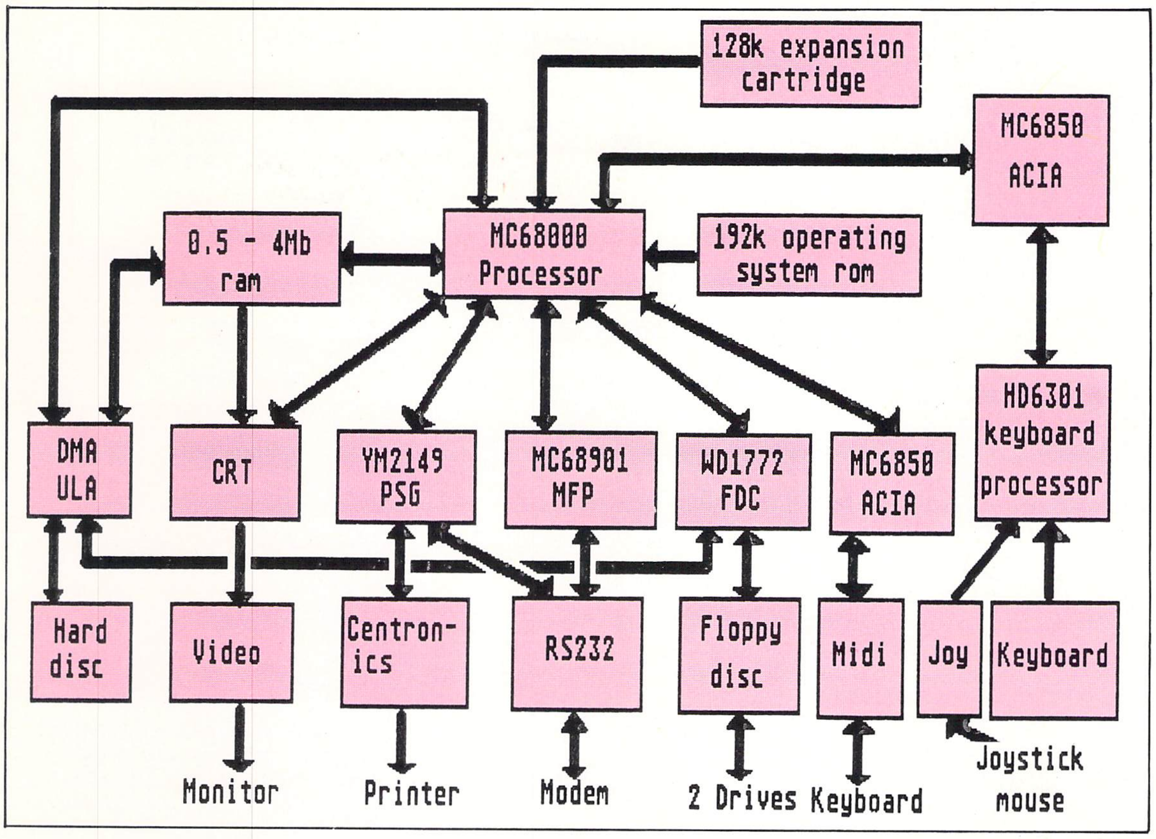
A	Keyboard – serial data transfer.
B	Cartridge connector.
C	DMA Port – hard disc, laser printer.
D	External floppy disc drive.
E	Monitor – RGB/monochrome only.
F	Midi in.
G	Midi out.
H	Centronics printer port.
I	240v AC mains socket.
J	RS232 serial port.
K	Power switch.
L	Cooling fan.
M	Reset button.

The ST's internal structure

SO far we have lifted the lids of the 520STFM and Mega 2 STs, looked inside and seen which components are used and where they are located on the main printed circuit board. We have also given a brief description of what the components are and their task within the ST.

We'll move on now to examine some of the important components in slightly more detail, and see where they fit in with the overall structure and design of this range of complex microcomputers.

Circuit and wiring diagrams of the PCB – printed circuit board – are extremely useful to the experienced hardware engineer, but for our purpose a simple block diagram will suffice to show how the components are connected and how they interact with each other.



The overall structure of the ST

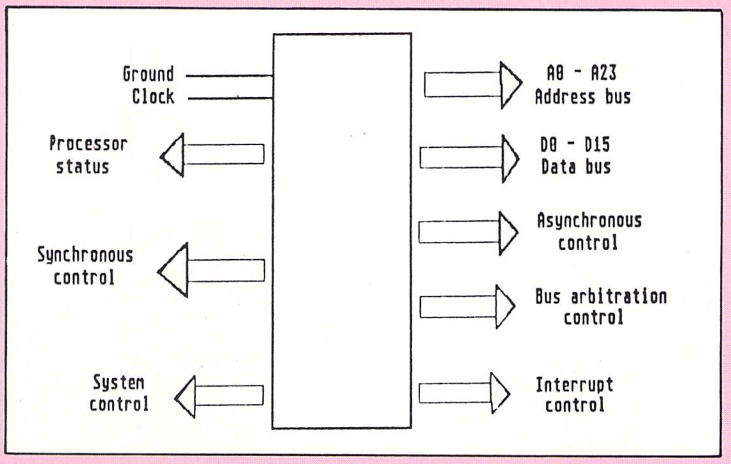
Motorola MC68000 microprocessor

The 68000 can run at a variety of speeds from around 4MHz up to about 12MHz. Atari chose a middle of the range frequency of 8MHz for maximum reliability.

It is a 16 bit processor with 32 bit internal architechture, and contains a total of 32 registers – eight 32 bit data, and seven 32 bit address registers, two 32 bit stack pointers and one 32 bit program counter.

The 24 bit address bus (pins A0 to A23) enables the processor to directly access 16 megabytes of memory. When accessing words of memory the odd address is treated as the low byte and the even address is the high byte, and you can only access words with an even address.

The 16 bit data bus enables byte and word sized chunks of data to be accessed. Each time memory or an I/O call is made the processor provides information on the status lines saying whether it is accessing data or program memory and in user or supervisor mode. The MMU – memory management unit – monitors these.



The 68000 processor – the heart of the ST

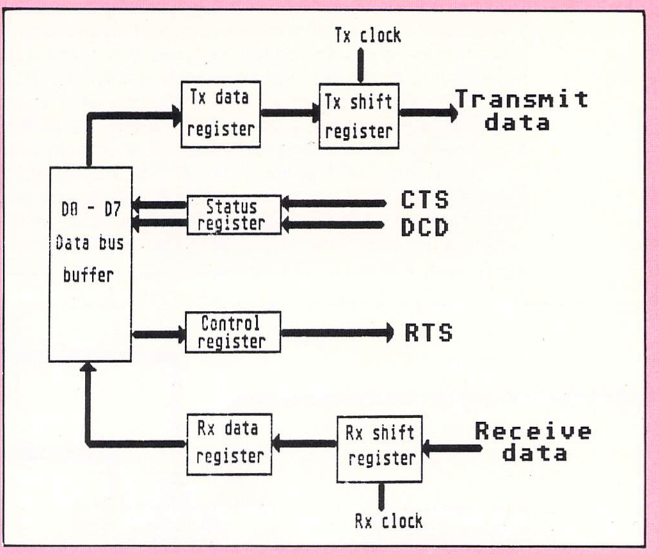
Bus arbitration control allows a peripheral device like the DMA – direct memory access – controller to take over operation of the data and address bus. Interrupt control provides eight levels of priority for peripherals requiring attention from the processor ranging from zero – no interrupt – to seven – a non-maskable interrupt.

The ST's internal structure

MC6850 asynchronous communications interface adapter

The ST uses two of these 24 pin ICs, one for serial communication with the mouse, keyboard and joysticks, and the other for the serial Midi port. The Midi port may be reconfigured as a second serial port, say for networking STs. The ACIA communicates with the 1MHz HD6301 8-bit keyboard controller at a fixed speed of 7.8Kbits per second.

The 6850 has four registers, two read only and two write only. The status and receive data, and the control and transmit data registers appear as two addresses in the ST's memory map. The keyboard ACIA is at $FFFC00 and the Midi interface is at $FFFC04.



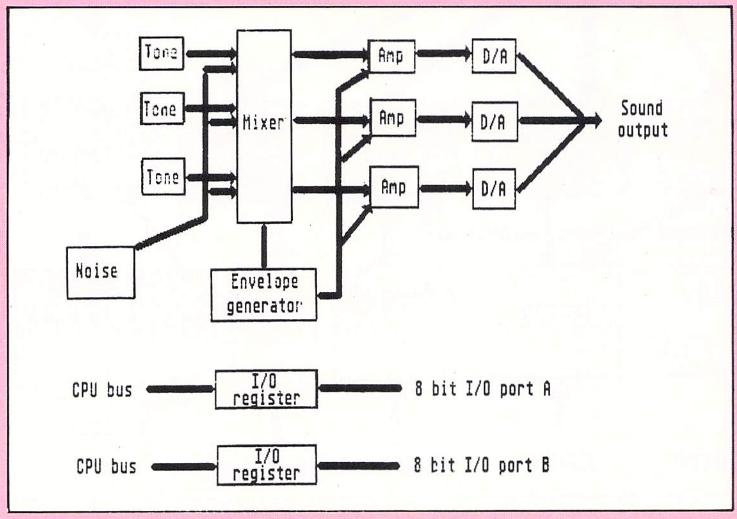
The MC6850 asynchronous communications interface adapter

Yamaha YM2149 programmable sound generator

This chip has three independently programmable pure tone generators, and a programmable noise generator combined with a mixer for the noise and tone channels. It has 15 volume levels and programmable envelopes complete with attack, decay sustain and release.

In addition to this it has a fully software controlled analogue output and two bi-directional 8 bit data ports.

The YM2149 PSG has 16 internal registers which can't be directly addressed – write to &FF8800 with the register number in the bottom four bits.



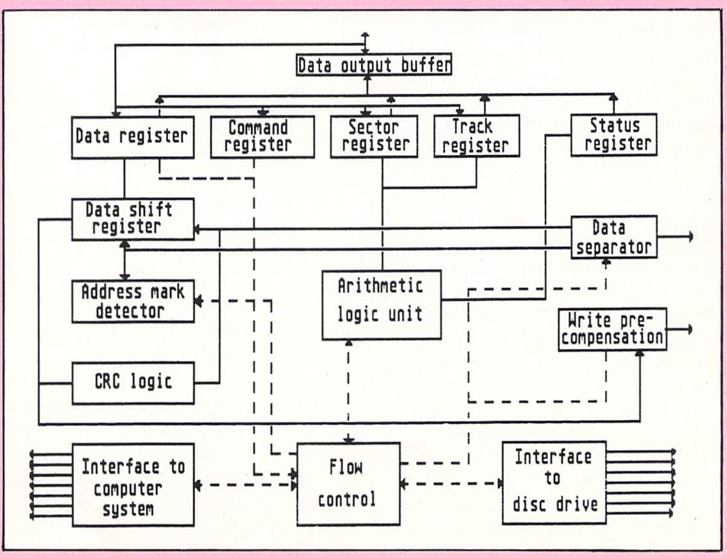
The structure of the sound system, incorporating the Yamaha YM2149 programmable sound generator

WD1772 floppy disc controller (FDC)

Developed by Western Digital, this chip contains all that is necessary for controlling 3.5in (and 5.25in) disc drives. It is capable of working in both double and single density modes, but in the ST is restricted to the former so the maximum amount of data is stored on each disc.

Sector lengths can be 128, 256, 512 or 1,024 bytes long – the ST uses 512 byte-sectors and the format is very similar to IBM's. It has a variable stepping rate – the time the FDC waits between tracks when moving the read/write head across the disc – from 2ms up to 6ms.

To avoid read errors, when each sector of data is written to the disc a 16 bit checksum (CRC – cyclic redundancy check) is calculated from the data and stored after it. When the sector is read back at a later date the checksum is recalculated and compared with the one on disc. If they both match then the read operation was successful.



The structure of the WD1772 floppy disc controller

Fault finding

THE ST is a very reliable micro and it should provide you with many years of trouble-free computing. Occasionally things do go wrong, but some of the faults are relatively simple and easily cured. It is also possible that when an ST appears to be faulty, the cause can be something completely different, like a damaged floppy disc. So let's look at some common problems and see how to isolate and cure the faults.

Power problems

When connecting up your ST and peripherals – monitor, hard disc, printer, modem, Midi keyboard, cartridge and so on – always make sure that the power to all the items is off. Plugging in peripherals when they are powered up can cause quite serious damage. And the same goes for unplugging them too – always remember to switch everything off first.

After plugging everything in insert a formatted disc – the UK Language disc provided with the micro will do – into the drive and first switch on all your peripherals, then your ST last of all. After a short while the desktop should appear.

What happens if it doesn't? The fault could lie almost anywhere, but we'll just look at some of the more common places. First, check that the power on light in the bottom left-hand corner of the keyboard is lit.

If it isn't the problem could lie with the fuse in the plug. Check this and replace it with a 3 Amp one – not 13 Amp as this could cause the ST a great deal of damage if a serious fault occurs. You could also test the power at the wall socket by plugging in a table lamp.

There's another fuse inside the ST on the power supply board. However, if this has blown it may indicate a more serious fault than the one we are trying to trace, and a qualified technician may be called for.

If the power light is on but you haven't got a picture of the desktop within a few seconds, check the monitor or television power light if it has one. If this isn't lit check the fuse. Make sure the brightness and contrast are turned up, and if using a TV, make sure it is tuned to the right channel.

If the power to the micro and monitor is on, but still no desktop try switching on the ST again, but this time without a disc in the drive. In this case the desktop should appear after about two or three minutes. If it does, the disc you tried to boot up with may be faulty. Test it by switching on with another disc that you know to be OK in the drive.

Printer problems

NORMALLY, when you hold down the Alternate key and press Help the screen will be dumped to the printer. It's useful for making hard copies of the screen when using software.

If nothing happens, first check that the printer is connected and on line. Again check that the fuse is OK. If still nothing happens, it may be that the software simply won't allow you to make a screen dump. This is true of most games software.

What commonly happens is that the printer puts a blank line in between each printed line and so ruins the dump. This is caused by the DIP switches being set incorrectly inside the printer. They should not be set to produce automatic paper feeds. The printer manual will tell you the location of the switches and how to alter them.

Disc problems

DRIVE faults aren't very common, and most errors can be traced to corrupted discs. Here are a few simple rules to observe when using discs; they will cut down on the number of errors:

- Don't switch off the micro when the drive busy light is on.
- Don't insert or remove discs when the busy light is on.
- Don't leave discs in the drive for long periods when they aren't being used.
- Don't leave discs in the sun.
- Don't leave them near magnets – for instance, on top of your hi fi loudspeakers.
- Don't touch the disc surface with your finger.

Many software packages are supplied on discs using a non-standard format so they can't be copied. If you try, you are liable to get disc errors or corrupted software that won't run – there's nothing wrong with your ST.

Mice problems

IF your mouse isn't working properly make sure it is plugged in correctly, and in the right port – port zero on the left. The mouse picks up dust and dirt quite easily and occasionally needs cleaning. Turn the mouse over and slide the cover over the ball in the direction of the arrows. The cover and ball should fall out.

The metal rollers inside the mouse body can be reached quite easily and gently wiped clean. The ball can be cleaned with a soft dry cloth.

Software problems

IT is quite rare to find a completely bug-free piece of software, and most packages have one or two undocumented features – bugs. If a program crashes or bombs out then the fault is either due to a hardware or software problem – always suspect the latter first.

Try switching off and running the package again, as the software may have become corrupted for some reason. Pressing the reset button clears the memory and resets the ST to the state it was in when it was switched on. However, some packages can survive a reset – particularly ram discs – and you may find that the next package you boot up won't run properly. Switching off for a few seconds will probably cure this.

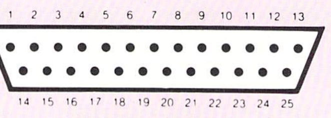
Finally, a word or two about viruses. They aren't too common on the ST, but there are one or two around, and they are liable to multiply if not stamped out.

Viruses are simply programs that are stored on a floppy disc, but can't be spotted either because they are stored on a special part of the disc called the boot sector, or because they are given filenames which imply they are something completely different, even useful.

They are loaded when the ST is booted up with a disc containing the virus, and like desk accessories, they are installed automatically. They can't damage your hardware, but they can do irreparable damage to your software. Switch off your ST to get rid of them and boot up with a disc you know to be free of the virus, like an unformatted one fresh from a new pack.

Pin-outs of the ST's ports

Modem

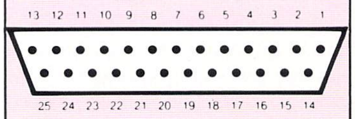


1	Protective ground	20	Data terminal ready
2	Transmitted data	21	Not connected
3	Received data	22	Ring indicator
4	Request to send	23-25	Not connected
5	Clear to send		
6	Not connected		
7	Signal ground		
8	Data carrier repeat		
9-19	Not connected		

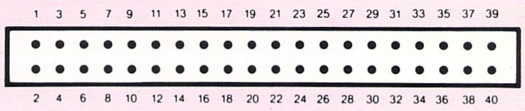
Fault finding

Printer

1	Strobe output
2	Data 0
3	Data 1
4	Data 2
5	Data 3
6	Data 4
7	Data 5
8	Data 6
9	Data 7
10	Not connected
11	Busy input
12-17	Not connected
18.25	Ground



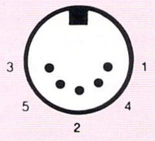
Cartridge



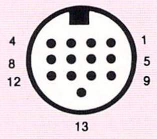
1	+5VDC
2	+5VDC
3	Data 14
4	Data 15
5	Data 12
6	Data 13
7	Data 10
8	Data 11
9	Data 8
10	Data 9
11	Data 6
12	Data 7
13	Data 4
14	Data 5
15	Data 2
16	Data 3
17	Data 0
18	Data 1
19	Address 13
20	Address 15
21	Address 8
22	Address 14
23	Address 7
24	Address 9
25	Address 6
26	Address 10
27	Address 5
28	Address 12
29	Address 11
30	Address 4
31	Rom select 3
32	Address 3
33	Rom select 4
34	Address 2
35	Upper data strobe
36	Address 1
37	Lower data strobe
38-40	Ground

Midi Out

1	Thru transmit data
2	Shield ground
3	Thru loop return
4	Out transmit data
5	Out loop return



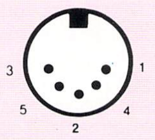
Monitor



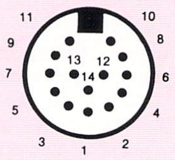
1	Audio out
2	Composite sync
3	General purpose output
4	Monochrome detect
5	Audio in
6	Green
7	Red
8	Plus 12-volt pullup
9	Horizontal sync
10	Blue
11	Monochrome
12	Vertical sync
13	Ground

Midi In

1	Not connected
2	Not connected
3	Not connected
4	In receive data
5	In loop return

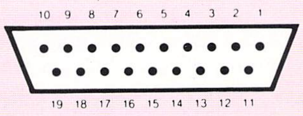


Floppy Disc



1	Read data
2	Side 0 select
3	Logic ground
4	Index pulse
5	Drive 0 select
6	Drive 1 select
7	Logic ground
8	Motor on
9	Direction in
10	Step
11	Write data
12	Write gate
13	Track 00
14	Write protect

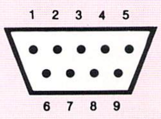
Hard Disc



1	Data 0
2	Data 1
3	Data 2
4	Data 3
5	Data 4
6	Data 5
7	Data 6
8	Data 7
9	Chip select
10	Interrupt request
11	Ground
12	Reset
13	Ground
14	Acknowledge
15	Ground
16	A1
17	Ground
18	Read/write
19	Data request

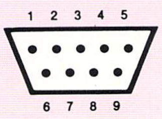
Mouse/Joystick

1	Up/XB
2	Down/XA
3	Left/YA
4	Right/YB
5	Not connected
6	Fire/left button
7	+5VDC
8	Ground
9	Joy1 fire/right button



Joystick

1	Up
2	Down
3	Left
4	Right
5	Reserved
6	Fire button
7	+5VDC
8	Ground
9	Not connected



Exploring the ST's drives

THE words disc and disc drive are often used when talking about computers, but how many people really know what makes the ST's disc drive work? By understanding the physical nature of discs and drives you will be more able to cope with the situation should things start to go wrong.

What makes them tick?

A DISC system is a fast and convenient way of storing programs and data. Large powerful mainframe computers – and some STs – use hard discs. These are made of rigid aluminium and require a very precise drive mechanism as well as clean air to operate in.

Most micros like the ST use flexible or floppy discs. These have a much smaller capacity for storing data and are much slower, but generally they are more than adequate for the single user, as they are much cheaper and require less finicky conditions than hard discs.

The first point to strike a newcomer when looking at a disc is the unmistakable fact that it is square. But a moment's inspection will reveal that inside the square plastic case is a round disc made of thin flexible plastic.

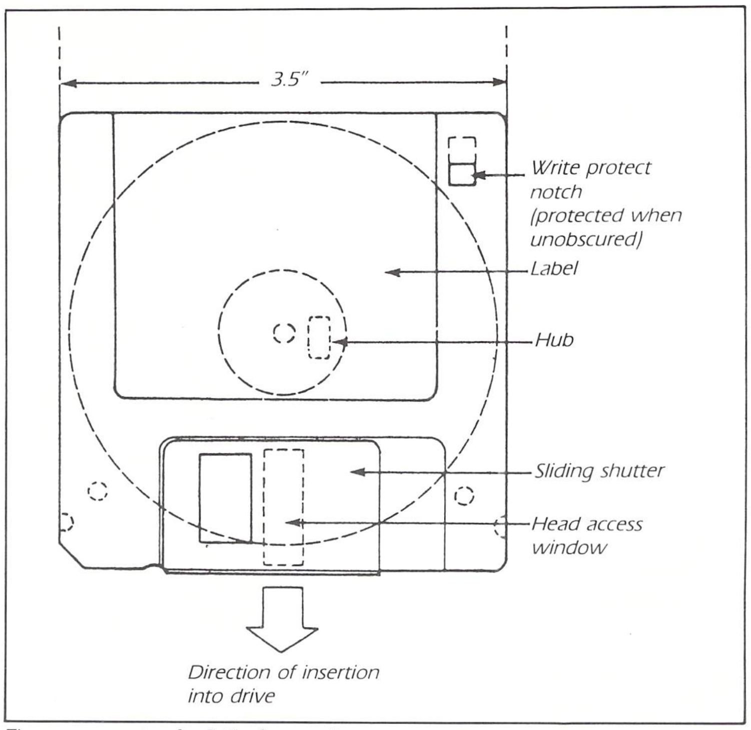
Coated on the plastic is a dark brown, sometimes black, layer of magnetic material. This is usually some form of metal oxide, ferric being the most common. The coating is like the one you get on music cassette tapes, so you can see that a disc is like a cross between a tape and a long playing record.

The track record

DO you remember the old riddle: "How many grooves are there on a long playing record?" The answer is, of course, two – one on the front and one on the back. For a floppy disc however, there are no physical grooves. Data is stored as a series of magnetic tracks – concentric rings of data. All tracks hold the same amount of data despite those on the outside being longer than those on the inside.

Some computers do have different amounts of data on different tracks, but most are like the ST and have the same. This simplifies the disc filing system, and as outer tracks tend to be more frequently used, it also slightly increases reliability. Some drives – like the ones built in to older 520STFMs – only use one side of the disc, but all new STs use both sides.

The disc drive is the "record player", a device to read and write data to the disc. Basically it consists of a drive motor to rotate the disc inside its plastic case. It does this by gripping the centre of the disc. Once it is spinning the read/write head is moved across the surface to position itself over the correct track.


3.5"
Write protect notch (protected when unobscured)
Label
Hub
Sliding shutter
Head access window
Direction of insertion into drive

The components of a 3.5in floppy disc

Floppy discs

SONY was the first company to produce the 3.5in disc system, the format used in the ST, and manufactured the first 3.5in drive to appear in quantity. It was designed to be an extension of the 5.25in drive currently in widespread use, and is completely compatible with it – meaning you can also use 5.25in drives with your ST.

The floppy disc is housed within a hard plastic case and has a spring loaded metal shutter which covers a slot where the read/write head accesses the disc. A single-sided disc has an unformatted capacity of 0.5Mb and a double-sided drive has a capacity of 1Mb.

A precision servo-controlled DC motor rotates the disc at 300rpm ±1.5%, and the drive must read each bit of data in 0.5 microseconds.

Moving ahead

THE head is usually moved by means of a stepping motor, which moves through a very small angle, usually 7.5 degrees, every time it receives a pulse. The rotational motion of the motor is converted into lateral head movement by a helical worm gear. Thus the head can be moved with precision over the surface of the disc.

However, when the drive is first switched on the head can be in any position, so there must be some way of finding precisely where it is. This is usually done by a small micro switch which is tripped by the head when it is over track zero.

The ST will keep issuing pulses to step the head backwards until this switch is tripped. This is known as restoring the head.

Magnetic disc storage

AT one time magnetic tape was the standard storage media for all types of computer systems, but this has been superseded by magnetic discs. Disc drives are available in a variety of sizes and forms, from the extremely high capacity multi-platter hard discs found in mainframes to the small 3.5in drives in the ST.

All disc drives are basically the same, despite differing outward appearances, and all operate in essentially the same manner. They record a pattern of binary numeric data in the magnetic oxide surface coating of the plastic disc. This pattern is written by a small coil contained within a tiny read/write head. The coil also acts as a mini magnetic receiver which can recognise the small magnetic pulses that represent data bits on the disc.

Unlike a hard disc drive, the floppy drive head actually touches the disc surface.

Exploring the ST's drives

Communicating via the bus

IN addition to the mechanical parts, a disc drive contains the electronics required to turn the motors on and off, as well as writing to the disc. These electronics communicate with the ST over the disc bus, which is just a series of signal wires used to connect several devices together.

Most disc drives stick to a standard bus layout on the edge connector at the back of the drive. So manufacturers can have their own design of drive electronics and still be compatible with other people's products.

As more than one drive can be connected to a bus, each one must be assigned a number. This is done by making a link on the electronics board. Each drive on the bus must have a unique number to prevent more than one drive being active at any one time. The select signals on the bus will therefore activate only one drive.

At the ST end of the disc bus there is a WD1772 disc controller, a very complex device. It accepts command numbers from the ST's 68000 microprocessor and generates the sequence of pulses on the disc bus to enable the drive to carry out the required action.

The 68000 can issue a command to move the read/write head to track 10. The disc controller chip then looks to see where the head is, and works out how many steps, and in what direction they will be, in order to get to it. It then issues that number of head step pulses.

Finally, when the head is in position it reads the track identification number to confirm that it is at the right one. Having completed that task it reports back to the microprocessor that the move has been made successfully.

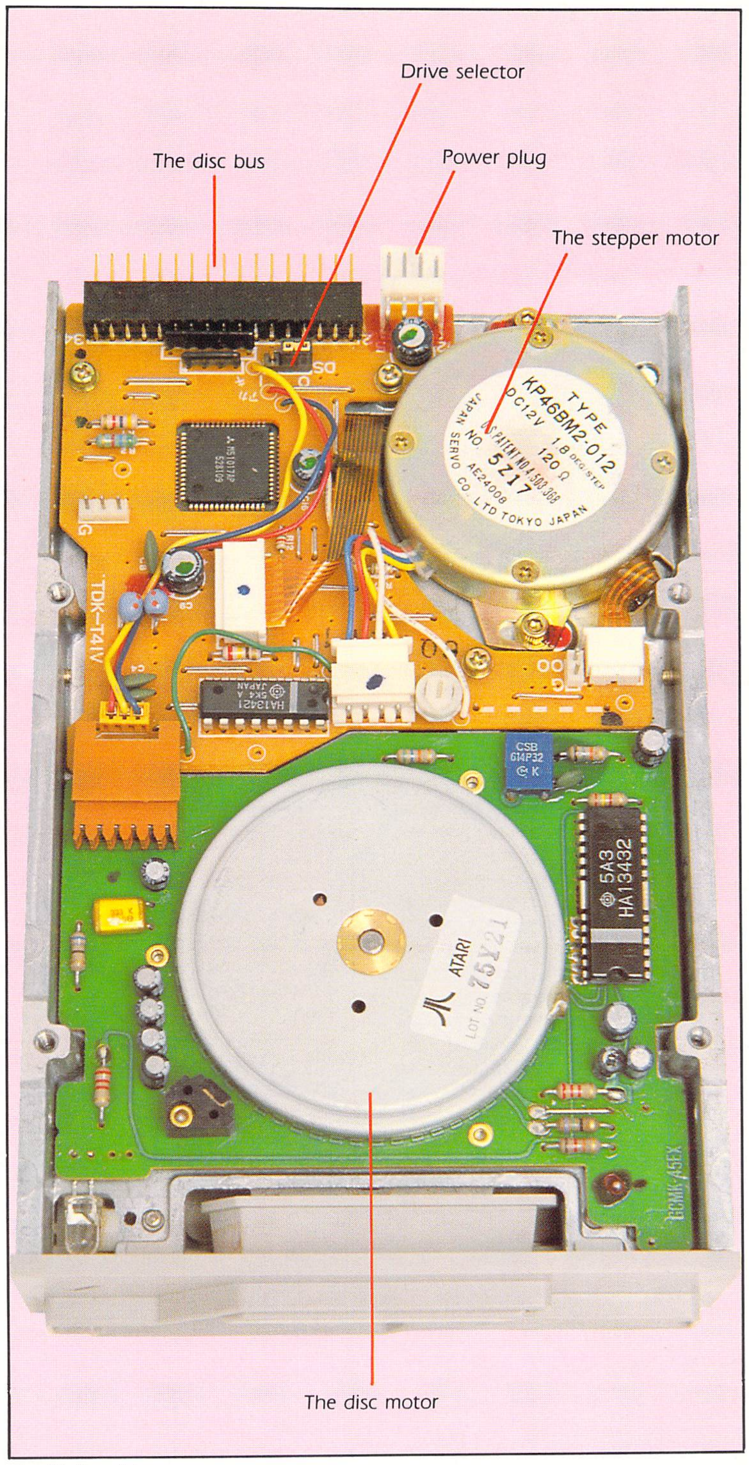
If the move was not a success this fact is reported and it is up to the disc filing system software to take appropriate action. Usually, the head is restored (moved to track zero) and another attempt is made. Several such attempts may be made before the disc filing system reports an error.

Chunks of data

THE track identification number mentioned earlier is put on to the disc during the formatting procedure which every disc has to go through before it can be used. This writes on the disc track and sector information.

We have already seen that a track is a ring of data stored on the disc, but this is still too large a chunk of storage to be convenient. This is because disc storage would have to be allocated in tracks, thus wasting a lot of space – a whole track would have to be written to save just one byte.

To remedy this, each track is broken down into a number of sectors – usually nine, but 10 is possible. A sector is the smallest unit of storage the disc holds, and all data transfer to and from the disc is done with sectors of data.



An ST disc drive

Hard discs

The ST's hard disc is a descendent of the first large multi-platter Winchester hard drives used on mainframe computers. These were very bulky and expensive compared to the ones available for the ST now. Originally 14 inches in diameter, they have been compacted to as little as 3.5 inches.

The disc rotates at around 3,000rpm – 10 times faster than a floppy – leading to rapid access times. Because of the vast amount of data crammed on to a hard disc it is made to a much greater degree of precision than a floppy drive. It is normally fully enclosed within its own dust-free environment, and a tiny read/write head the thickness of a human hair floats on a thin cushion of air a fraction of an inch above the disc surface.

The drive is quite susceptible to damage because of the small distance between the disc surface and the head. Jolting it can cause the head to crash against the disc surface with disastrous consequences.

Getting set up

The object of this section is to investigate the ways that a small business can be successfully managed or operated using an ST. We aim to look at how to begin your set-up, the various pitfalls to avoid and some of the best software available to get the most from your ST in business.

Equipping yourself for the job

WHETHER we like it or not, we are immersed in Margaret Thatcher's share-owning and small business democracy. Since 1984, more than 340,000 small businesses have started up using the government's Enterprise Allowance Scheme. This provides for an individual who can invest a minimum of £1,000 in their chosen concern and supplements the business at a rate of £40 a week for the first year. Many other small companies have started under their own steam without government help.

It is perhaps coincidental that the ensuing four years have also seen the most startling growth in the use of home micros and PCs. It is therefore not surprising that as a result of the technological revolution more and more businesses are reliant for their day-to-day running on the power of the silicon chip.

The vast majority of these new small businesses are being run by people who have become self-employed for the first time, and perhaps more significantly, have had little or no experience with micros or PCs.

Business people are on the whole concerned only with the value of such technology to the growth and prosperity of their business. But it is easy to become lost in a world of techno-jargon and to make totally inappropriate use of these most powerful pieces of equipment. Worse than that, it is also quite common for budding entrepreneurs to spend hard earned money on a computer system and then let it gather dust on a desk, doing no work whatsoever.

The ST is a versatile micro which is well supported by some excellent business software. Both the 520STFM and 1040ST retail at less than £500 and can provide a firm foundation for a computer-run small business at minimum outlay.

It doesn't matter whether your business is that of a rural hairdresser, local greengrocer, wholesale garden gate supplier, architectural survey service, personal tutoring concern or even a software distributor, the ST can provide the important infrastructure needed for your enterprise to succeed.

The set-up

THE first difficulty is encountered in defining what constitutes a small business. Sole trading or any business which employs fewer than 20 employees surely falls into the category of small, but exactly what the ceiling may be is debatable.

Most people starting self employment for the first time will begin – at least initially – as a one man operator (sole trader) or by employing only a handful of people. Consequently, an ST would make an ideal first purchase.

For most purposes a 1040ST with a monitor would be a wiser investment than a 520ST, for the justifiable reason of the extra memory capacity for applications such as databases and DTP packages. At a later date it will be a simple process to upgrade to a Mega ST. However, if you expect rapid expansion it is probably more cost effective to go straight for a Mega 4 ST.

Second disc drives at this point are not a necessity but are very useful to speed up copying of data between discs. A modem, allowing your ST to send fax, telex and Email messages, is becoming more and more essential for companies of all sizes.

For most business uses a reasonable dot matrix printer is a minimum requirement. Panasonic, Epson and Brother all produce such printers at affordable prices.

The choice between a hi-res monochrome or medium-res colour monitor is one of intrinsic choice and size of wallet – remember that you will pay three times more for a colour monitor. Some business packages will run only in monochrome, but increasingly software is being released which runs in medium or low resolution, and as such a colour monitor should not place too many limitations.

A solid desk or table situated at a judicious position in your premises is essential as a workstation to site your prized micro. An expensive plinth is not necessary for your monitor, as viewing angle is a matter for personal preference. You can always construct a DIY unit from chipboard to mount the monitor at eye level.

Alternatively, modern research shows that the optimum position for a VDU is below the level of the micro – as per television newsreaders. As this would involve cutting a large chunk out of your desk, it is not a strategy I would recommend.

Thankfully your printer can be placed in any position that the lead allows and where paper feed is not snarled. All that matters is that you are comfortable working with your ST and monitor. If however, backache or eyestrain result after a longish working session, I would suggest some alteration of the set-up arrangement or height and position of your seating.

Getting to grips

If this is your first meeting with an Atari ST it is certainly a wise ploy to play with your new micro for a few days before attempting to use it for business purposes. Use some of the introductory software and don't feel embarrassed if you spend half your time at this stage playing space invaders. Such play and experimentation will help you get a feel for your ST and also aid you in coming to terms with some of its eccentricities.

Read the user's manual which comes with your micro and learn by heart the important procedures of copying, formatting, write protecting, booting programs, opening and closing windows and using folders – especially preparing AUTO folders. Also practice setting preferences, utilising the control panel, saving the desktop and generally getting used to working in the Gem environment. In short, get to know your ST before taking any unnecessary risks with business data.

Good habits

Ensure that you make a working copy of your ST Language disc then store the original upright in a safe, cool and dry place. Follow this policy with any new commercial disc you may buy – providing of course that the software allows back-up.

Wherever possible work with a back-up disc and keep your valuable masters out of harm's way. You never know when you might format or wipe a disc in error so it is always prudent to have copies. As long as you don't distribute them to friends you will not be breaking any piracy laws.

Another habit which is worth forming early on is to keep commonly used discs close to hand in a top drawer or in a rack or box on the desk top. As your collection increases,

Getting set up

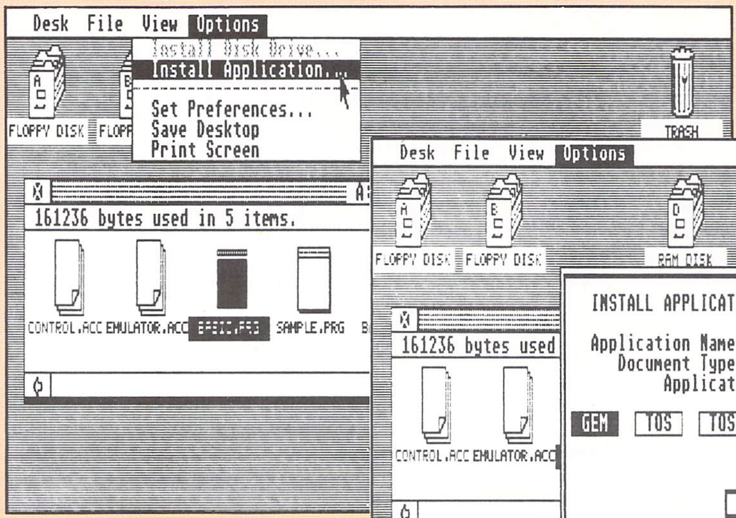
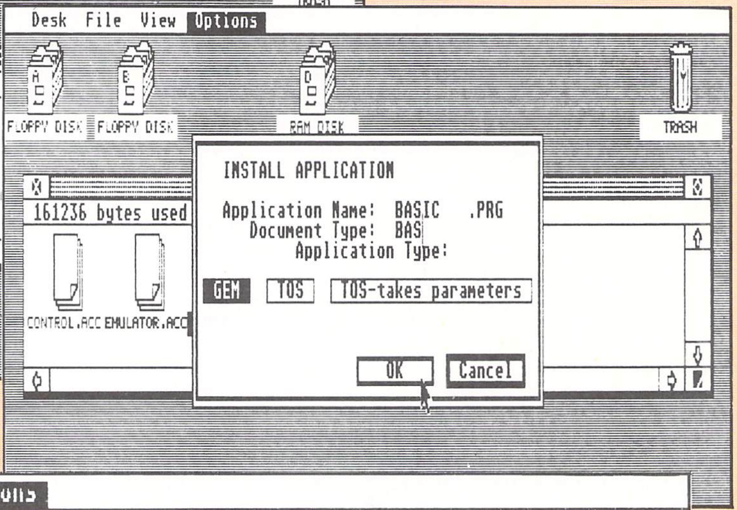


Figure I: To install an application upon boot up choose Install Application from the Options menu



INSTALL APPLICATION

Application Name: BASIC .PRG
Document Type: BAS
Application Type:

[✓ GEM] TOS TOS-takes parameters

OK Cancel

Figure II: File type = BAS

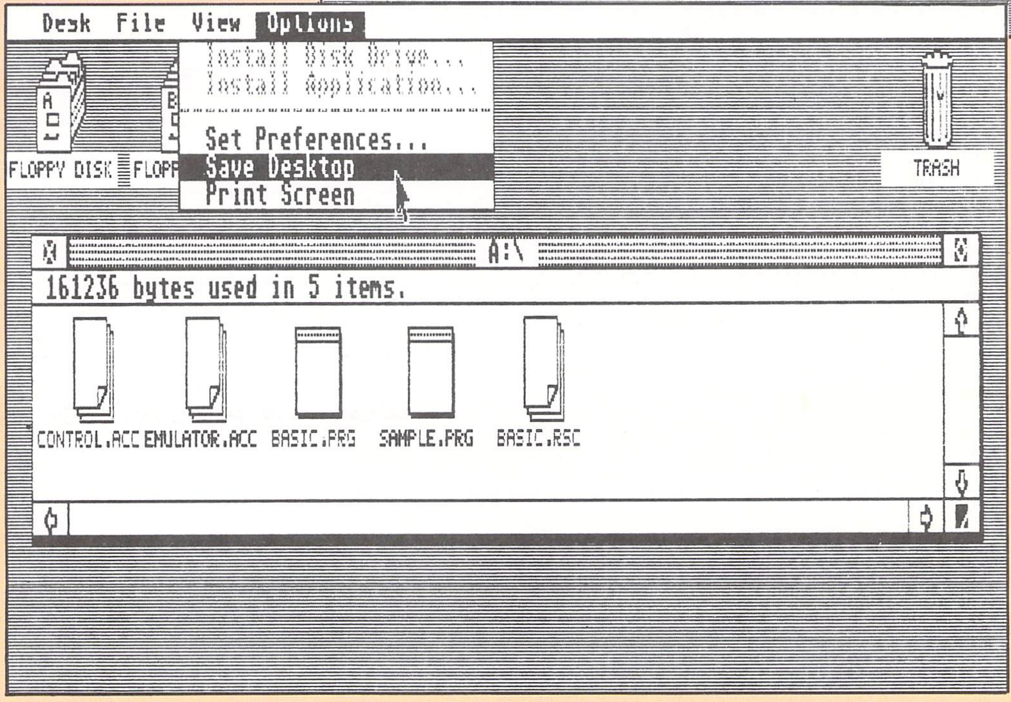


Figure III: Finally save desktop

ensure that you index them for fast and easy access. There can be nothing more frustrating than scrambling in search of important data at crucial moments in the life of your business.

It is perhaps a sagacious idea to switch on your ST at the beginning of each day and to leave it switched on until close of business in the evening. A micro uses very few units of electricity compared to most household appliances, so you have no need to worry about suddenly soaring electricity bills. This way you will have immediate access to any important file or letter during your working day without the nuisance of having to power-up all of your computing equipment.

A useful process to learn at the start of your undertaking is a quick routine for installing some business applications. It can be cumbersome to have to load an application and then load previously saved data. It is much easier to double click on the data file, have the application boot up and the data file automatically load itself.

This can be achieved using this working example: It is possible to alter the desktop so that double-clicking on a .BAS file causes ST Basic to load with the clicked file already in it.

If you assume that ST Basic is the application you wish to install, boot up the desktop and open a directory window of the ST Basic disc. Single click on the BASIC.PRG file so it goes black, then pull down the Options menu and select Install Application.

You will be asked for the document type. At this stage enter BAS. and click on OK. Again bring down the Options menu, but this time opt for Save Desktop. Now by double clicking on any file with a .BAS ending ST Basic will automatically load and the BAS file of your choice will be inserted ready to run or edit. As you saved the desktop, you will be able to do this whenever you boot up this Basic disc. Equally, of course, this method can be transferred to any type of application and will work with many database, spreadsheet and word processor packages.

However you must select the appropriate file type for the application you are dealing with. This may take some practice so once again it becomes part of your spare time learning activities.

It is also helpful to install a ram disc in an AUTO folder on your work disc, which will be loaded when you boot up the disc. This will enable fast copying and flexible storage capacity when working with large amounts of data.

After a period you will perhaps surprise yourself with the amount of use you gain from a fast ram disc.

Many good Ram disc programs are available in the public domain, either to be downloaded through MicroLink or on 3.5in discs from PD libraries. These cost little more than the price of the disc.

Getting into word processing

IN this section, examining how the ST can be put to use in the office and small company, we will look at the many different packages available – word processors, spreadsheets and databases – that can streamline a business and improve its efficiency.

First we'll look at word processors. The cost of these range from free public domain packages up to ones costing several hundred pounds. The questions a businessman must ask is: "Which package is best for me?", "What can a £200 word processor do that a £10 one can't?".

To enable you to decide which package provides you with the right features at the right price we'll take a whistle-stop tour through what's available, starting with the free programs.

Experimenting with one of these will enable you to make a more informed choice of what suits you best.

You may find everything you want, or you may say: "If only it would do this ... or that ...". Armed with this knowledge you can go and buy the package that actually has the features you need.

Word processors

A WORD processor is a piece of software which will enable you to write letters, forms, memos and articles on screen and works rather like an electronic typewriter.

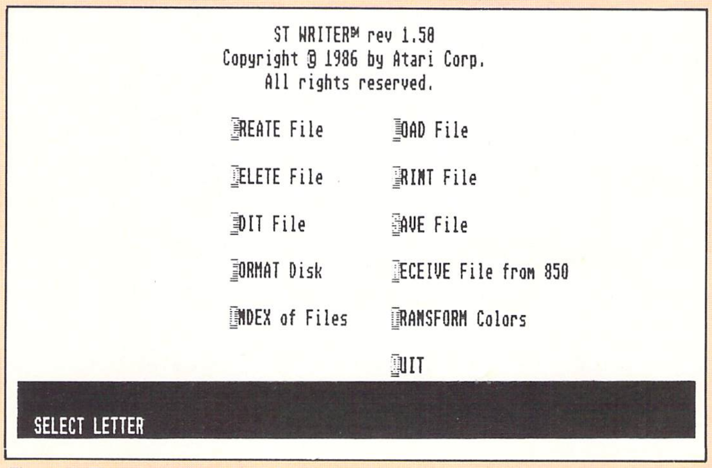
These can then be changed, edited and have their fontstyle altered at will without resort to correction fluid, scissors or paste. They can then be saved to disc or exported to a printer and duplicated freely on A4 or foolscap paper, envelopes or labels.

For anyone familiar with an electronic typewriter the transition to a word processor should be quite painless. The advantages over more traditional methods of writing soon become obvious.

In the world of these hi-tec writing aids, ST Writer, Speedwriter and 1st Word are obtainable from PD libraries – you will normally only have to pay for duplication costs and the price of a blank disc. Each are superb user-friendly word processors, with full on-screen help documents and are capable of handling large letters and mail-shots.

1st Word is Gem-based and fully wysiwyg – which means: What you see on screen is what you get on paper – with easy to use editing facilities. Conversely, ST Writer and Speedwriter are extremely fast but non-wysiwyg. All are worth looking at and may help colour your preferences for future word processing.

However, none of these programs include spelling checkers nor mail-merge facilities available on commercial contemporaries. These extra additions along with utilities such as Etikette – a label printing program – can be collected from the public domain.

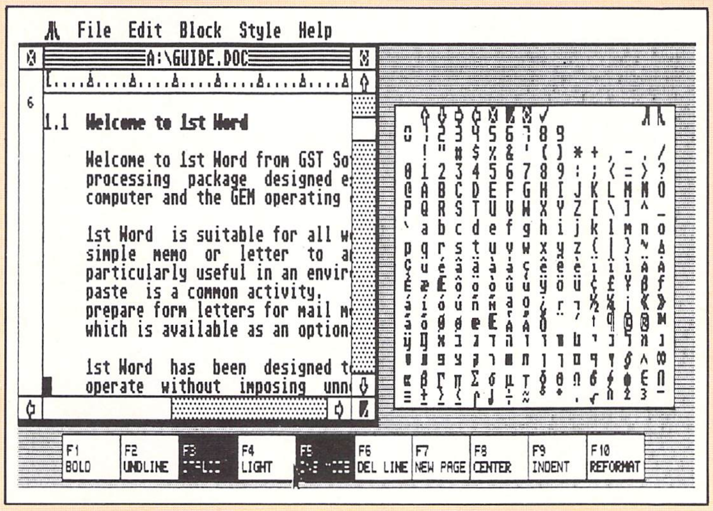


ST WRITER™ rev 1.58
Copyright © 1986 by Atari Corp.
All rights reserved.

CREATE File	LOAD File
DELETE File	PRINT File
EDIT File	SAVE File
FORMAT Disk	RECEIVE File from 850
INDEX of Files	TRANSFORM Colors
	QUIT

SELECT LETTER

Figure I: The command page from ST Writer



1.1 Welcome to 1st Word

Welcome to 1st Word from GST So
processing package designed
computer and the GEM operating

1st Word is suitable for all w
simple memo or letter to a
particularly useful in an envir
paste is a common activity.
prepare form letters for mail m
which is available as an option

1st Word has been designed t
operate without imposing unn

Figure II: 1st Word's edit page and character set – note the wysiwyg display

A closer look

1st Word, and its contemporaries which are Gem-based use the mouse to select commands. Edit screen commands which can be accessed by pressing any one of a series of pre-defined function keys are available on most types of word processor. These can do anything from deleting lines of text to reformatting blocks.

Keyboard driven word processors load up with what is called a command page as can be seen in Figure I. This is the command page from ST Writer and allows you to select from 11 options. Other packages present a similar selection.

On the command page you will usually begin by selecting Create File – or document – and the screen will then toggle to what is called the edit page. However, on Gem-based versions the screen will load in edit mode with function commands available on screen in a separate box.

It is on the edit page that you will write the text you wish to process. You can return to the command page at any time by pressing the Escape key. This is true with most word processors.

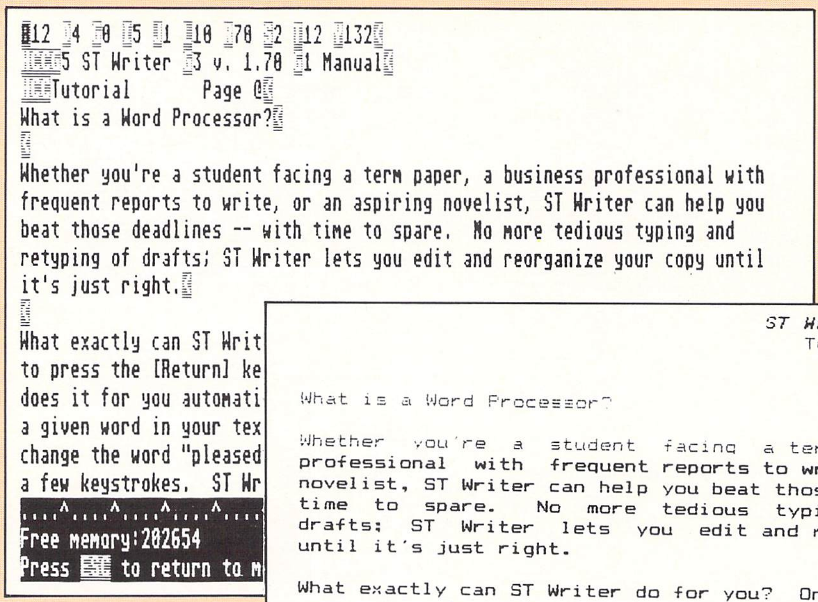
On the edit page you will be able to set-up vertical and horizontal rulers to justify your text in preparation for printing it out. This is normally achieved by setting headers.

You type in the header code – as explained in the individual help documents – and then set a value for it. For instance T0 would set the margin at the top of your page to zero and L7 would give seven spaces in the left margin of the page.

Word processors also allow you to define a ruler – that is the width of text you wish to create. This is achieved either by selecting a command such as Default Ruler from the command or edit page. If you bear in mind that you would normally print out to a default of 40 or 80 characters width you can redefine the ruler to anything from 5 to 80.

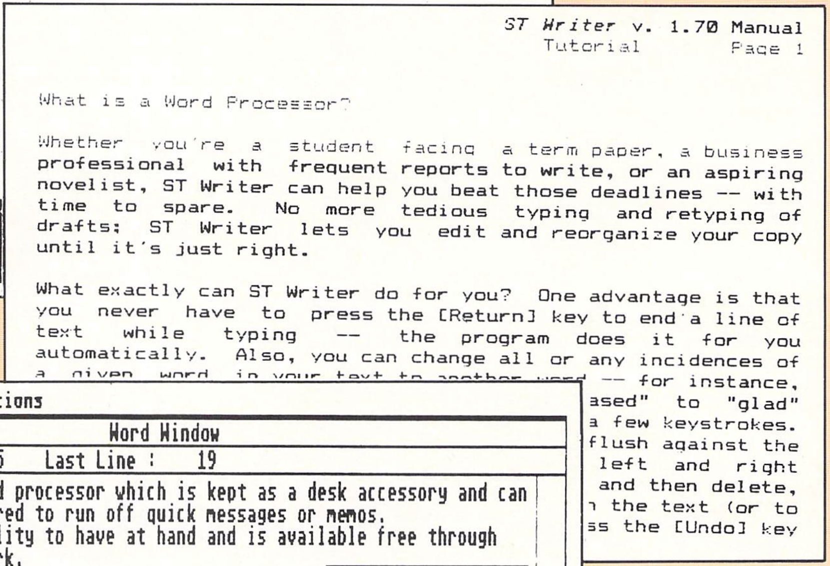
It is best to experiment with these headers and rulers before you attempt

Getting into word processing



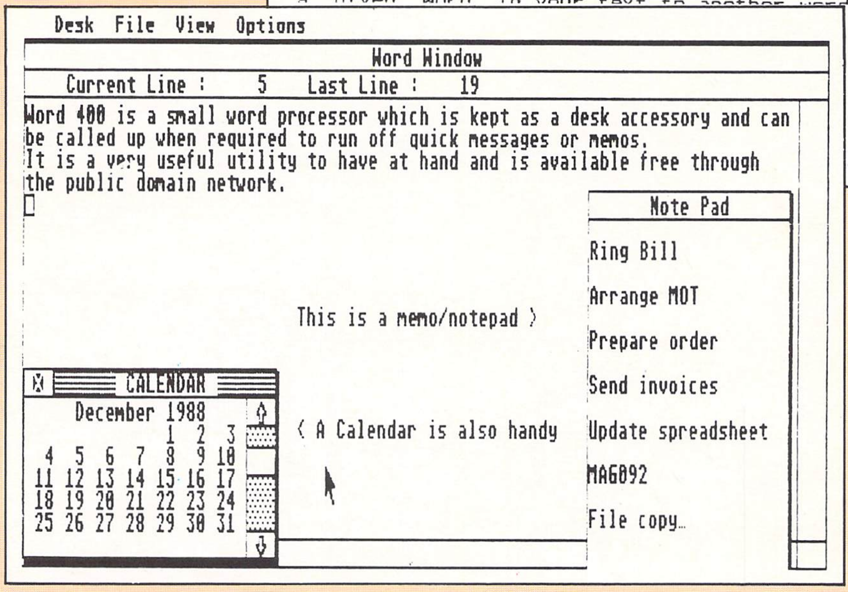
```
12 4 8 5 1 10 78 2 12 132
5 ST Writer 3 v. 1.70 1 Manual
Tutorial        Page @
What is a Word Processor?

Whether you're a student facing a term paper, a business professional with
frequent reports to write, or an aspiring novelist, ST Writer can help you
beat those deadlines -- with time to spare.  No more tedious typing and
retyping of drafts; ST Writer lets you edit and reorganize your copy until
it's just right.

What exactly can ST Writ
to press the [Return] ke
does it for you automati
a given word in your tex
change the word "pleased
a few keystrokes.  ST Wr
....^....^....^....^....
Free memory:202654
Press ESC to return to m
```

Figure III: St Writer's non-wysiwyg edit page



```
                                   ST Writer v. 1.70 Manual
                                     Tutorial          Page 1

What is a Word Processor?

Whether you're a student facing a term paper, a business
professional with frequent reports to write, or an aspiring
novelist, ST Writer can help you beat those deadlines -- with
time to spare.  No more tedious typing and retyping of
drafts;  ST Writer lets you edit and reorganize your copy
until it's just right.

What exactly can ST Writer do for you?  One advantage is that
you never have to press the [Return] key to end a line of
text while typing -- the program does it for you
automatically.  Also, you can change all or any incidences of
a given word in your text to another word -- for instance,
                                      ased" to "glad"
                                      a few keystrokes.
                                      flush against the
                                       left and right
                                       and then delete,
                                      the text (or to
                                      ss the [Undo] key
```

Figure IV: Print out from ST Writer



```
Desk  File  View  Options
                    Word Window
  Current Line :     5    Last Line :    19
Word 400 is a small word processor which is kept as a desk accessory and can
be called up when required to run off quick messages or memos.
It is a very useful utility to have at hand and is available free through
the public domain network.

                                                    Note Pad
                                                 Ring Bill
                                                 Arrange MOT
                   This is a memo/notepad >      Prepare order
                                                 Send invoices
        CALENDAR
      December 1988                              Update spreadsheet
               1  2  3  < A Calendar is also handy
   4  5  6  7  8  9 10                           MAG092
  11 12 13 14 15 16 17
  18 19 20 21 22 23 24                           File copy...
  25 26 27 28 29 30 31
```

Figure V: A selection of useful desk accessories

to write an important document. However, most word processors also have a set of default value margins and ruler and these are usually adequate for many purposes.

Many word processors also allow you to change the on-screen and print-out style. These can be carried over to the printed document via the default printer driver.

Non-Epson printers may need another printer driver, so you should check this out first before buying your selected printer. An example of a 1st Word character set and part of the edit screen can be seen in Figure II.

Typical on-screen and print-out views of work produced by a word processor are shown in Figures III and IV.

Other functions

All good word processors allow you to search and replace letters and words in a selective or global fashion – that is, individual or all occurrences. This is normally achieved by setting up the replace option on the command page or via a command option on the edit page. This utility is particularly useful for editing and correcting text already prepared.

Another similar facility is the block movement option which lets you mark blocks of text and move them to any position in your complete script. In most cases you will need to set markers at the beginning and end of the block you wish to move.

This is achieved by moving the screen cursor to the desired position before clicking on a defined key or icon. Once again this gives a flexibility to your writing which cannot be achieved with a normal electronic typewriter.

Many small but useful programs prove invaluable to the novice businessman while he is word processing. These include the aforementioned spelling checkers, word counters and mailmerge facilities, but it is also handy to have the current monthly calendar and correct time on-screen during the working day.

Calendar and clock programs proliferate in both the public domain and commercial market. They are usually available as desk accessories which are loaded upon boot up.

Also useful are note pad and word window programs which allow you to keep memos and make jottings on screen – some such utilities contain alarms which sound for important appointments. These memos can either be saved to disc or printed out for reference.

Figure V shows a typical screen containing a word window, note pad and calendar. This can easily be toggled to desktop or overlaid on a working document.

The next step in word processing

PUBLIC domain word processors for the ST are powerful utilities which outperform by several orders of magnitude their contemporaries on the 8 bit micros. However, the commercial equivalents are even more breathtaking, and despite their often high price tags will repay the small businessman by automating many aspects of his day to day work, thus saving time and money. Let's look at what the market leaders have to offer.

The quid pro quo

THE best commercial word processors for the ST contain the same standard editing aspects as featured in their PD contemporaries – such as search and replace – plus a lot more besides. All run in 80 column mode on colour medium resolution or monochrome high resolution monitors, but working in colour on a TV will cause severe eyestrain, and is best avoided.

Each permits variation in font style, whether this is limited to normal, italic, bold and underline type as in K-Word 2 or a more comprehensive selection available in HB Marketing's Word Up. Equally, unlike 1st Word and ST Writer, the commercial alternatives also have in-built wordcounters. These are invaluable when you need to write to a set total word length.

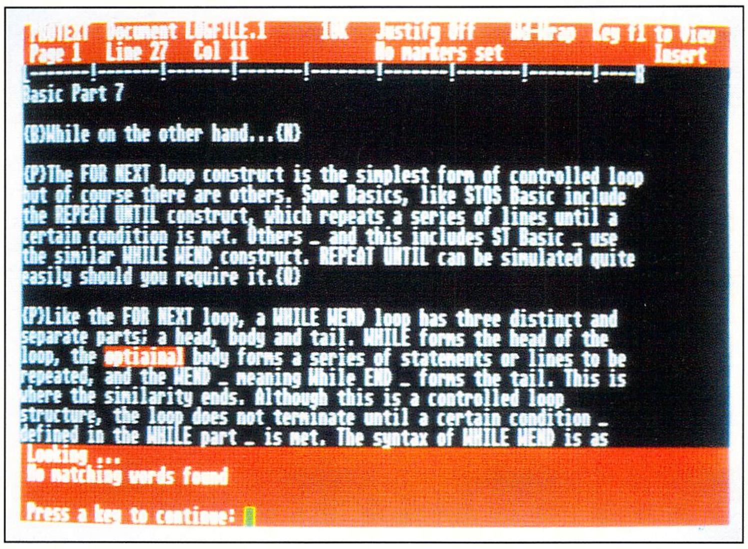
All allow selective page numbering and some include a master page which can be customised with details such as a heading or chapter number and set as a default for each page you type.

TextPro, Word Perfect and Word Writer ST also have an index facility. This enables you to index large pieces of work to ease cross-referencing and quick searching for important facts and figures.

Unique additions also include 1st Word Plus' Spill File which enables use of a hard disc for overflow of memory – particularly useful if you need to write book-length pieces of text.

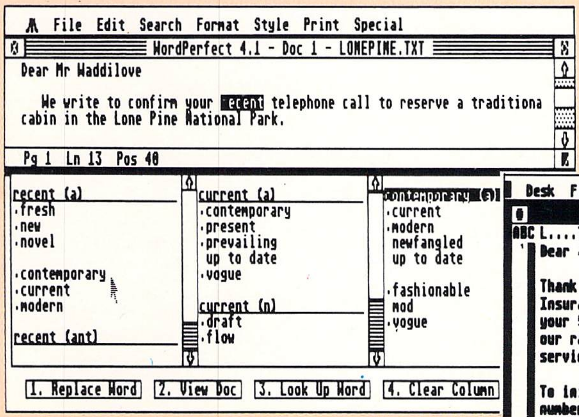
Other special features are included in the table below which covers seven of the top word processors for the ST. Though new products appear on the market all the time and updates are part of an ongoing process, these titles have a proven pedigree.

Basically it's a matter of selecting an affordable piece of software, looking at what it has to offer and then trying it out for yourself.



```
PROTEXT  Document LONFILE.1        10K   Justify Off    Wd-Wrap  Key f1 to View
Page 1   Line 27   Col 11                No markers set                  Insert
[--------!--------!--------!--------!--------!--------!--------!--------!----R
Basic Part 7

(B)While on the other hand...(B)

(P)The FOR NEXT loop construct is the simplest form of controlled loop
but of course there are others. Some Basics, like STOS Basic include
the REPEAT UNTIL construct, which repeats a series of lines until a
certain condition is met. Others _ and this includes ST Basic _ use
the similar WHILE WEND construct. REPEAT UNTIL can be simulated quite
easily should you require it.(B)

(P)Like the FOR NEXT loop, a WHILE WEND loop has three distinct and
separate parts; a head, body and tail. WHILE forms the head of the
loop, the optiainal body forms a series of statements or lines to be
repeated, and the WEND _ meaning While END _ forms the tail. This is
where the similarity ends. Although this is a controlled loop
structure, the loop does not terminate until a certain condition _
defined in the WHILE part _ is met. The syntax of WHILE WEND is as
Looking ...
No matching words found

Press a key to continue:
```

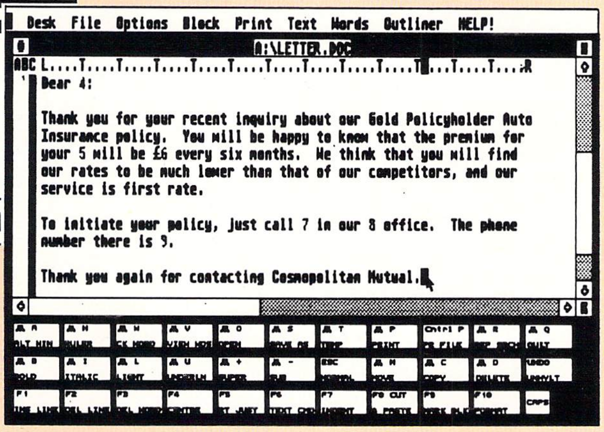
Protext spell-checking a document



```
WordPerfect 4.1 - Doc 1 - LONEPINE.TXT
Dear Mr Waddilove

   We write to confirm your recent telephone call to reserve a traditiona
cabin in the Lone Pine National Park.

Pg 1  Ln 13  Pos 40
```

recent (a)	current (a)	contemporary (a)
·fresh	·contemporary	·current
·new	·present	·modern
·novel	·prevailing	newfangled
	up to date	up to date
·contemporary	·vogue	
·current		·fashionable
·modern	current (n)	mod
	·draft	·vogue
recent (ant)	·flow	

Word Perfect's thesaurus in operation



```
A:\LETTER.DOC
ABC L....T....T....T....T....T....T....T....T....T....T....T....T...R
Dear 4:

Thank you for your recent inquiry about our Gold Policyholder Auto
Insurance policy.  You will be happy to know that the premium for
your 5 will be £6 every six months.  We think that you will find
our rates to be much lower than that of our competitors, and our
service is first rate.

To initiate your policy, just call 7 in our 8 office.  The phone
number there is 9.

Thank you again for contacting Cosmopolitan Mutual.
```

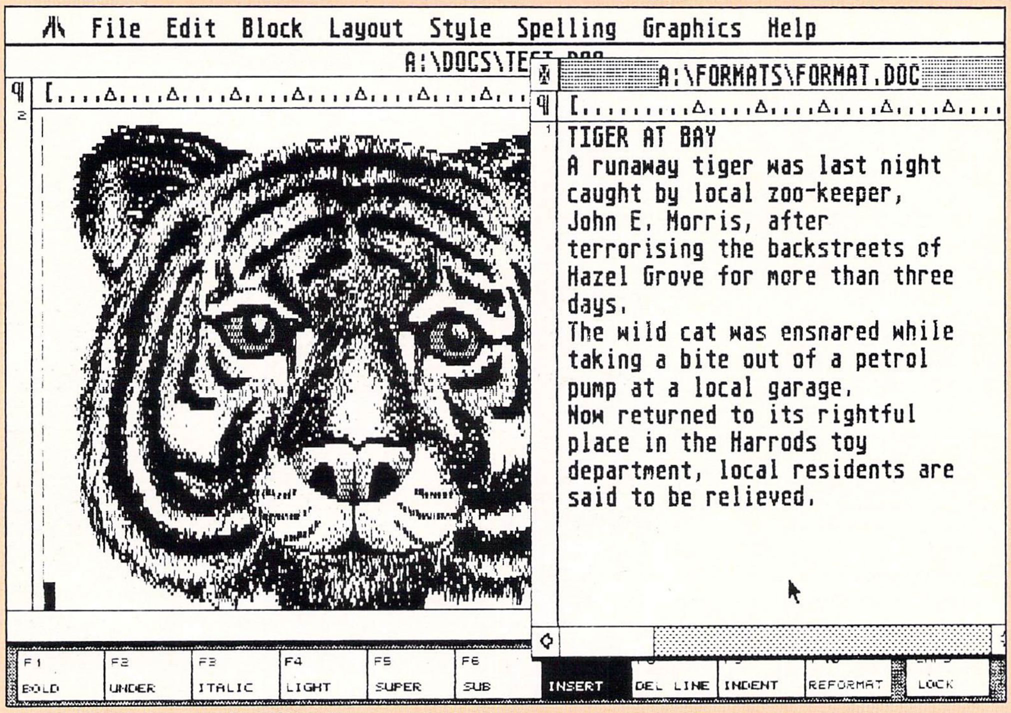
Word Writer ST showing a form letter for mail merging

Wordprocessor	Supplier	Price	Wysiwyg	Operation	Spell-check	Thesaurus	Mail merge	Preview	Microjustification	Pseudo DTP
TextPro	Precision	£39.95	No	Gem	No	No	Yes	Yes	No	No
Word Up	HB Marketing	£59.95	Yes	Gem	No	No	Yes	No	No	Yes
K-Word 2	Kuma	£59.95	Yes	Gem	Yes	*	Yes	No	No	No
1st Word Plus	GST	£79.95	Yes	Gem	Yes	No	Yes	No	No	Yes
Word Writer ST	Timeworks	£79.95	Yes	Gem	Yes	Yes	Yes	No	No	No
Protext	Arnor	£99.95	Yes	Non-Gem	Yes	No	Yes	Yes	Yes	No
Word Perfect	Sentinel	£228.85	Yes	Both	Yes	Yes	Yes	Yes	Yes	Yes

(* Available as a separate desk accessory)

The facilities offered by the top word processors for the ST

The next step in word processing



TIGER AT BAY
A runaway tiger was last night caught by local zoo-keeper, John E. Morris, after terrorising the backstreets of Hazel Grove for more than three days.
The wild cat was ensnared while taking a bite out of a petrol pump at a local garage.
Now returned to its rightful place in the Harrods toy department, local residents are said to be relieved.

The pseudo DTP style of 1st Word Plus

Common wordprocessor features

Wysiwyg: This determines whether the screen display matches the output you will get on your printout. Such a display gives an easier indication that what you are typing is what you really want to achieve – something which is not available on dedicated stand-alone word processors.

In all cases – with the exception of Protext – this is standard on the ST, and such displays may or may not show all margins, headers and footers. However, even Protext has the facility to toggle to a wysiwyg-type screen display.

Operation: Most software is primarily operated via the mouse, making use of an on-screen menu to quickly select various options – though typing is of course still via the keyboard. The style of this operation varies enormously from partial Gem loyalty – which can make word processing a little clumsy – to full implementation of Gem.

While Protext makes no concessions to this mode of working and as such is similar in use to a stand-alone word processor – Word Perfect allows both an orthodox keyboard approach and a flexible Gem usage.

Spell-check: This refers to the inclusion of a spelling checker dictionary which will selectively or globally search and correct spellings in your text. These facilities may be available within the program or loadable from a separate disc. If separate, then the dictionary is often larger, though spell checking may be slower than with a smaller glossary stored in the word processor program.

The size of these dictionaries varies, and includes 1st Word Plus' 40,000 word tome and Protext's hefty 66,000 which is more than adequate for most small business purposes. In many cases these dictionaries may be supplemented as new words crop up. For instance, 1st Word Plus and Protext allow user-created supplementary dictionaries – particularly useful if you wish to store spellings of commonly used technical, foreign or medical terms. Protext also includes a quick checker for the 10,000 most commonly used words, which is fast and important for checking spelling as you type.

Thesaurus: A few packages also boast a thesaurus which will aid searching for synonyms. This is particularly handy if writing is the core of your concern or if you need to express yourself lucidly.

Once again this addition normally has to be loaded from a separate disc before a selective search and replace can be put into operation.

Mail merge: This almost universal facility allows you to quickly merge data from spreadsheets and database programs into your text for many purposes. By transferring names and addresses from a database it also permits the production of macros – or form letters – to mailshot customers and suppliers. Each letter can also include its own personalised unique message, address and name.

This speedy process negates the need to write the same letter to dozens or even hundreds of clients, and is a professional time-saving boon to all progressive businesses.

Preview: This indicates a facility to allow you to see accurately on screen what you will receive on your final printout. It is particularly helpful with the non-wysiwyg display of Protext, and is also a useful addition on other word processors such as TextPro as it gives a clearer indication of the finished product than a straightforward wysiwyg display.

Microjustification: A most advanced addition which will create perfectly balanced lines when right justification is chosen. It creates microspacing between letters, allows for varying sizes of characters and alters spaces to create an almost typeset feel to your work.

Pseudo DTP: If home produced fly ads or graphically enhanced mailshots are called for, some word processors can emulate a pseudo desktop publishing environment. They do this by allowing inclusion of home drawn or imported graphic images into your text.

Such an environment will also allow you to columnise your text to produce a newsprint type layout – hence the term desktop publishing. Though these pseudo facilities are useful for many business concerns, none possesses the power and flexibility of a dedicated DTP package.

Keeping account

IF you run your own business, no matter how large or small, you will at one point or another have to deal with your accounts. Whether you simply keep a day book, produce a cash flow for the bank manager or provide end-of-year ledgers, a micro will smooth and speed the process.

As you probably know, accounts and accountants are expensive and time consuming. Using an ST and a powerful spreadsheet and accounting package will help you with these often tiresome procedures and also enable you to run your business more smoothly and efficiently.

There is a bewildering variety of packages on the market, and some are very expensive, so we'll take a brief tour of what's available and the features you can expect them to contain.

What is a spreadsheet?

YOU are probably familiar with ledger sheets used by accountants for making balance sheets and profit and loss statements. Well, a spreadsheet is an electronic tabulated worksheet which can be tailored to produce and monitor cash flows, depreciation charts, end of year ledgers and much more besides.

Not only is it capable of cross calculation of cash data, but it can also be used to classify and manipulate non-monetary arithmetic data, such as test results of students, IQ scores, cubic metres of earth dug each day by a gang of workmen and so on. Figure I shows an example.

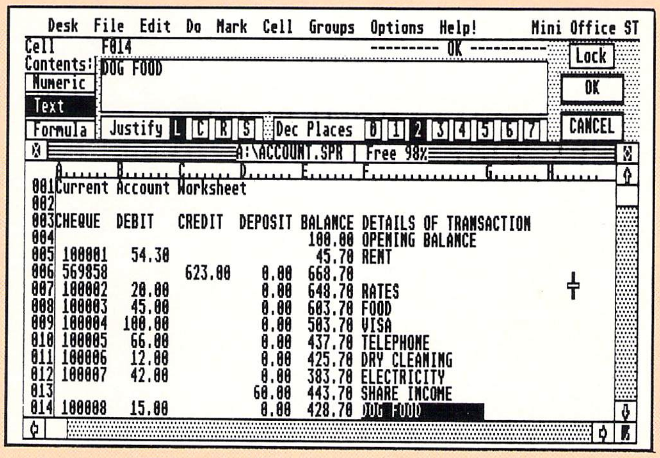
A spreadsheet is made up of a large number of cells – boxes – organised in rows and columns. These can contain numbers, text or calculations – in fact anything you care to put in them.

An average commercial spreadsheet program – which may allow up to 65,500 rows by 65,500 columns – can cater for 4,000,000,000 cells. That's if your ST had the memory to store them. The fact that you would never need to use anywhere near that amount of cell space is irrelevant, but it proves the sheer power of a typical package of this sort.

The joy of electronic spreadsheets to the businessman is their speed and ability to answer "What if?" questions. They can be constantly updated and altered – usually using an Edit command – without resort to writing out each sheet from scratch.

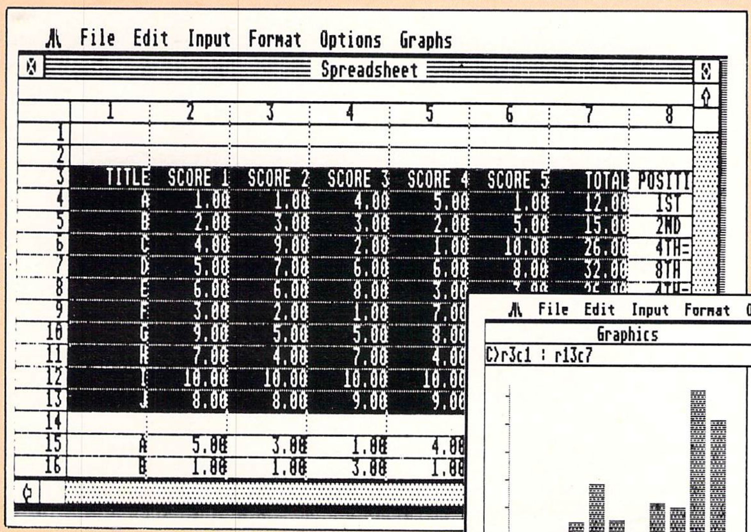
When one of the cells is changed it is reflected in the state of the others, so it is easy to predict the outcome of circumstances like increasing sales, overdrawn bank balances, cash flow problems and so on. Basically the sheet can be manipulated to provide the set-up of information which is required for your business.

Further to this, their global calculatory functions enable you to quickly do tasks which with a pocket calculator would eat away at valuable business time. They are also cost saving, as they let you carry out many of the weekly and monthly accountancy tasks which you would otherwise have to pay an accountant to do.


Desk File Edit Do Mark Cell Groups Options Help! Mini Office ST
Lock
OK
CANCEL
Numeric
Text
Formula
Justify L C R S
Dec Places 0 1 2 3 4 5 6 7

```
Cell      F014                          --------- OK ---------
Contents: DOG FOOD
                       A:\ACCOUNT.SPR | Free 98%
   A.....  B.....  C.....  D.....  E.....  F............  G.....  H.....
001Current Account Worksheet
002
003CHEQUE   DEBIT   CREDIT  DEPOSIT BALANCE DETAILS OF TRANSACTION
004                                  100.00 OPENING BALANCE
005 100001   54.30                    45.70 RENT
006 569858          623.00     0.00  668.70
007 100002   20.00             0.00  648.70 RATES
008 100003   45.00             0.00  603.70 FOOD
009 100004  100.00             0.00  503.70 VISA
010 100005   66.00             0.00  437.70 TELEPHONE
011 100006   12.00             0.00  425.70 DRY CLEANING
012 100007   42.00             0.00  383.70 ELECTRICITY
013                           60.00  443.70 SHARE INCOME
014 100008   15.00             0.00  428.70 DOG FOOD
```

Figure II: Mini Office Professional – a good Gem-based spreadsheet



```
        1        2        3        4        5        6        7       8
 1
 2
 3    TITLE  SCORE 1  SCORE 2  SCORE 3  SCORE 4  SCORE 5    TOTAL POSITI
 4        A     1.00     1.00     4.00     5.00     1.00    12.00   1ST
 5        B     2.00     3.00     3.00     2.00     5.00    15.00   2ND
 6        C     4.00     9.00     2.00     1.00    10.00    26.00   4TH=
 7        D     5.00     7.00     6.00     6.00     8.00    32.00   8TH
 8        E     6.00     6.00     8.00     3.00
 9        F     3.00     2.00     1.00     7.00
10        G     9.00     5.00     5.00     8.00
11        H     7.00     4.00     7.00     4.00
12        I    10.00    10.00    10.00    10.00
13        J     8.00     8.00     9.00     9.00
14
15        A     5.00     3.00     1.00     4.00
16        B     1.00     1.00     3.00     1.00
```

Figure I: Tabulation of test results is easy with a spreadsheet such as Glentop's Graphic sheet

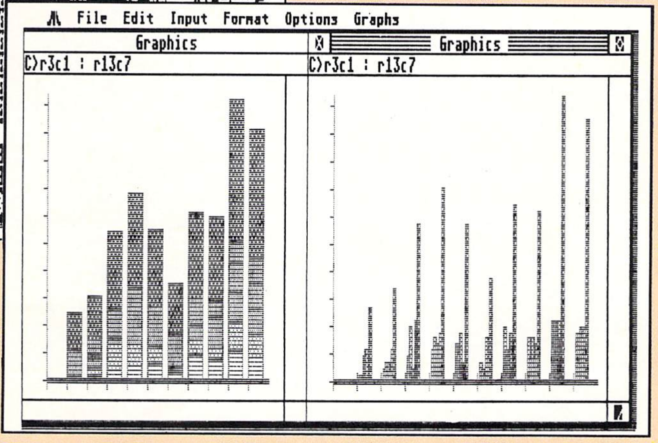


Figure III: And you can output the results in chart form

Keeping account

What's available

IN addition to varying speed and size, these packages can differ in many ways. One is whether they are Gem-based or command-driven. Mini Office Professional Spreadsheet, shown in Figure II, is a good example of the Gem-based variety. The pros and cons of operation reflect the same arguments in the word processor section, but command-driven sheets are rare for the ST.

If you are unfamiliar with computerised spreadsheets a Gem-based one will be easier to get to grips with than a command-driven one. However, once you get to know a command-driven one it can be much faster and flexible.

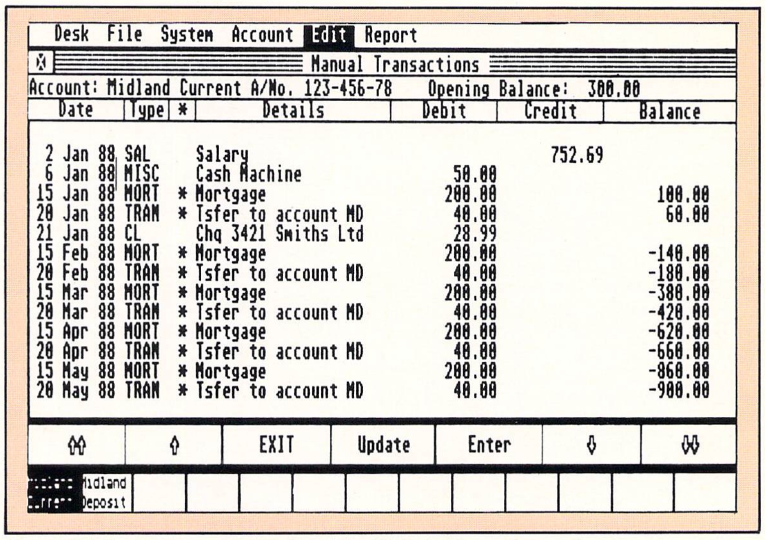
The second difference is whether or not they support graphics to allow you to print out graphs and charts from your accounts or statistics. These are handy as they can give at-a-glance updates on your commercial progress or forecasts. Glentop's Graphic Sheet provides some excellent graphic functions, as can be seen in Figure III.

Some spreadsheets also provide facilities for on-screen calculators and notepads to enable you to perform related tasks while updating your sheet.

In the public domain, spreadsheets and accountancy programs are as bearish as their word processing relations, but by necessity of their size are limited in their power compared to their more expensive big brothers. VC Spreadsheet – available on Softville's ACC.23 disc – is a marvellous non-Gem but highly usuable spreadsheet.

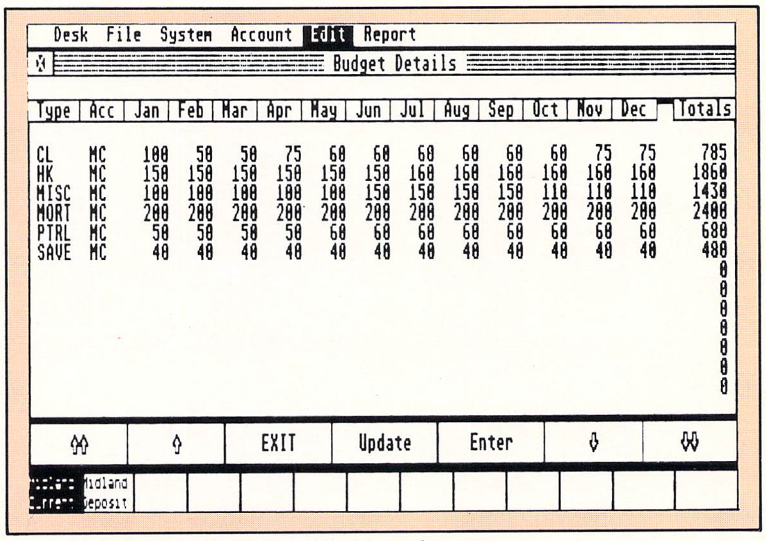
Meanwhile ST-Sheet, available on Softville's ACC.31 disc, is a Fast Basic spreadsheet utility that can be used as a stand alone program or as a desk accessory. Also available in the PD market is a superb accounts/spread utility to be used within the ST Writer environment.

If you require slightly more flexibility and power for ledgering and day to day accountancy, Kuma's K-Spread and Digita International's Digicalc are inexpensive and user-friendly entry level spreadsheets which hold a half-way position between the PD sheets and the lucrative master packages.



```
Manual Transactions
Account: Midland Current A/No. 123-456-78   Opening Balance:  300.00
   Date    Type *        Details          Debit     Credit    Balance

 2 Jan 88 SAL    Salary                             752.69
 6 Jan 88 MISC   Cash Machine              50.00
15 Jan 88 MORT * Mortgage                 200.00               100.00
20 Jan 88 TRAN * Tsfer to account MD       40.00                60.00
21 Jan 88 CL     Chq 3421 Smiths Ltd       28.99
15 Feb 88 MORT * Mortgage                 200.00              -140.00
20 Feb 88 TRAN * Tsfer to account MD       40.00              -180.00
15 Mar 88 MORT * Mortgage                 200.00              -380.00
20 Mar 88 TRAN * Tsfer to account MD       40.00              -420.00
15 Apr 88 MORT * Mortgage                 200.00              -620.00
20 Apr 88 TRAN * Tsfer to account MD       40.00              -660.00
15 May 88 MORT * Mortgage                 200.00              -860.00
20 May 88 TRAN * Tsfer to account MD       40.00              -900.00
```

Figure IV: Bank statement produced by Digita's Home Accounts



```
Budget Details

Type  Acc  Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec   Totals

CL    MC   100   50   50   75   60   60   60   60   60   60   75   75     785
HK    MC   150  150  150  150  150  150  160  160  160  160  160  160    1860
MISC  MC   100  100  100  100  100  150  150  150  150  110  110  110    1430
MORT  MC   200  200  200  200  200  200  200  200  200  200  200  200    2400
PTRL  MC    50   50   50   50   60   60   60   60   60   60   60   60     680
SAVE  MC    40   40   40   40   40   40   40   40   40   40   40   40     480
                                                                           0
                                                                           0
                                                                           0
                                                                           0
                                                                           0
                                                                           0
                                                                           0
                                                                           0
```

Figure V: Budget details produced by Home Accounts

Also available are some excellent general accountancy packages such as Microdeal's Personal Finance Manager and Digita's Home Accounts. These allow you to construct, tabulate and print many everyday accountancy tasks, such as bank statements – shown in Figure IV – and budget details – shown in Figure V.

At the top end of the market the packages are many and varied. VIP Professional is a very powerful but expensive Lotus 1-2-3 clone – the standard spreadsheet by which all others are compared in the IBM PC business world.

It is therefore best perhaps to begin with either a public domain or cheap spreadsheet then to decide for yourself what you require of a master package. The table summarises a cross-section of the best packages available.

Guide to Spreadsheet and Accountancy packages for the ST

Spreadsheet	Supplier	Price	Graphics mode	Operation	Package type
VC Spreadsheet	Softville	PD	No	Non-Gem	Simple spreadsheet
ST-Sheet	Softville	PD	No	Gem	Simple spreadsheet
K-Spread 2	Kuma	£59.95	Yes	Gem	Spreadsheet
Digicalc	Digita	£39.95	No	Non-Gem	Spreadsheet
Graphic Sheet	Glentop	£49.95	Yes	Gem	Spreadsheet
Mini Office Spreadsheet	Database	£29.95	No	Gem	Spreadsheet
Home Accounts	Digita	£29.95	Yes	Gem	General accounts
Swiftcalc	Timeworks	£79.95	Yes	Gem	Spreadsheet
Personal Finance Manager	Microdeal	£29.95	Yes	Gem	General accounts
BAS Accounts	BAS	£195.00	Yes	Gem	General accounts
Master Plan	Silica	£89.95	Yes	Gem	Spreadsheet
VIP Professional	Silica	£228.85	Yes	Gem	Spreadsheet

Getting into spreadsheets

A simple price list

SO far we have looked at the basic principles of spreadsheets and how they can prove invaluable for small business purposes. Now we will take the next important step and observe in more depth how you can make these electronic accounts sheets work for you in given situations.

Most spreadsheets for the ST work in similar ways, but they all have their own quirks and exclusive peculiarities. To explain some simple operations and calculatory functions we will use Glentop's Graphic Sheet and Kuma's K-Spread 3 as our working examples of spreadsheets in action.

To give you an idea of the saving in time – time is money in business – and effort that a spreadsheet and ST can bring, we'll work through a couple of simple examples.

Any ST spreadsheet will allow you to construct a price list like that shown in Figure I, instantly to calculate VAT and mark-up which can be constantly updated. Bear in mind however, that the actual mechanics for doing so will vary slightly depending on which sheet you use. For simplicity we will temporarily ignore VAT and look at how, using K-Spread 3, the price list was made up.

The first stage is to boot up the spreadsheet which should load with the edit cursor – the small oblong box – located in the top left hand corner of the sheet (cell A:0). Now your first tasks are fairly simple as you enter the column headings along row number one, pressing the right arrow after each heading has been entered.

However, you will notice that the heading Part Number partly disappears as you enter the next heading Title. This is because the first label was too long for the default width of the cell, so you will need to widen cell A:0.

This is a simple procedure: Locate the mouse arrow on the bottom right hand corner of A:0 and press the Alt and left mouse keys simultaneously. Now drag the arrow towards the right, release the key and the cell will have been widened to accommodate the words Part Number.

Now you are ready to enter the part numbers. To save typing time, sequential part numbers can be generated automatically using a FILL command in K-Spread 3's Options Menu.

To do this, select Fill Range from the Options menu. Choose the range of cells that you wish to fill by entering the parameters A1:A10 into the Range Dialogue box – shown in Figure II.

Now click on OK and Select Fill by Column by clicking on the downwards facing arrows. Choose the starting and finishing points of the part numbers, in this case 1001 and 1010, then select an increment of one. This procedure can be clearly seen in Figure II. The incremental part numbers will now be displayed in column A, rows 1-10.

The next important step is to set up a template for generating the price list. You will see at first hand the calculatory speed and power of an ST spreadsheet as it automatically works out selling prices as cost prices are entered.

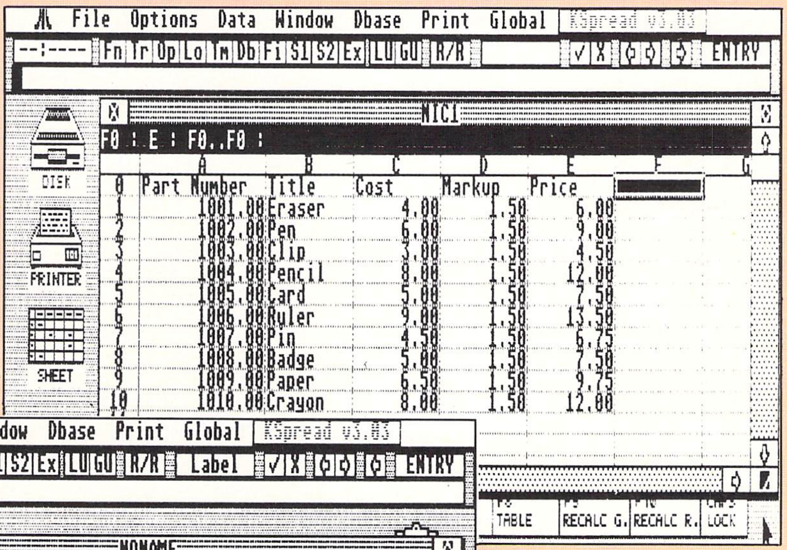
The percentage mark-up should be entered into cell D1. For this example we assume the mark-up to be 50 per cent, so 1.50 – one whole plus 50 per cent – is entered into cell D1. The mark-up is standard across the product range, so this figure can be copied down the mark-up column.

To do this, select cell D1 by clicking on it and then drag it to the column header. Enter into the Range Dialogue the parameters 1:10 and press Return. The markup 1.50 will now be copied to the relevant range.

Now you must set up the formula to perform the calculations, the object being that the spreadsheet automatically works out the selling price in column E by multiplying the cost by the mark-up.

Next enter some sample prices into column C1 and enter the calculatory formula C1*D1 (cost times mark-up) into cell E1.

The formula can be copied down the sheet by selecting and dragging the cell onto the column header, entering

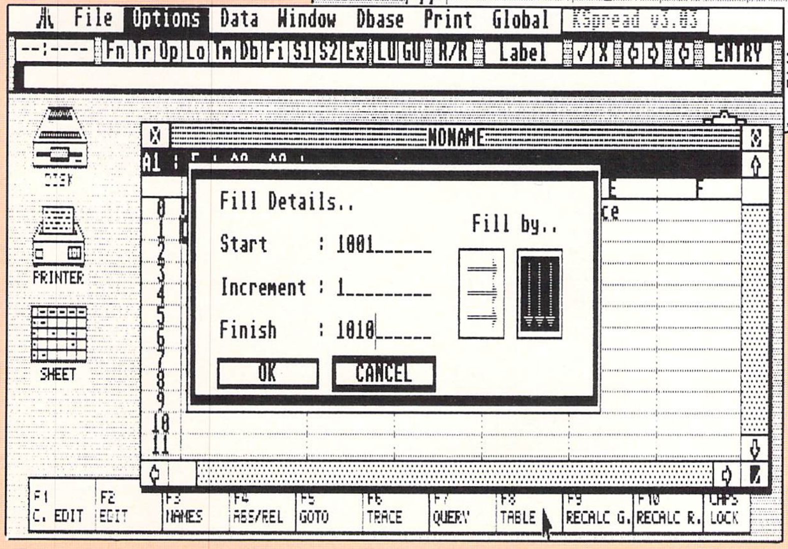

File Options Data Window Dbase Print Global KSpread v3.03
--:---- Fn Tr Op Lo Tm Db Fi S1 S2 Ex LU GU R/R ✓ X ENTRY
DISK
PRINTER
SHEET
F8 TABLE F9 RECALC G. F10 RECALC R. CAPS LOCK

NIC1

F0 : E : F0..F0 :

	A	B	C	D	E
0	Part Number	Title	Cost	Markup	Price
1	1001.00	Eraser	4.00	1.50	6.00
2	1002.00	Pen	6.00	1.50	9.00
3	1003.00	Clip	3.00	1.50	4.50
4	1004.00	Pencil	8.00	1.50	12.00
5	1005.00	Card	5.00	1.50	7.50
6	1006.00	Ruler	9.00	1.50	13.50
7	1007.00	Pin	4.50	1.50	6.75
8	1008.00	Badge	5.00	1.50	7.50
9	1009.00	Paper	6.50	1.50	9.75
10	1010.00	Crayon	8.00	1.50	12.00

Figure I: A simple price list created using Kuma's K-Spread 3


File Options Data Window Dbase Print Global KSpread v3.03
--:---- Fn Tr Op Lo Tm Db Fi S1 S2 Ex LU GU R/R Label ✓ X ENTRY
DISK
PRINTER
SHEET
NONAME
F1 C. EDIT F2 EDIT F3 NAMES F4 ABS/REL F5 GOTO F6 TRACE F7 QUERY F8 TABLE F9 RECALC G. F10 RECALC R. CAPS LOCK

Fill Details..

Start : 1001

Increment : 1

Finish : 1010

Fill by..

OK CANCEL

Figure II: The Range Dialogue box

Getting into spreadsheets

the range as before and pressing Return. The formula will automatically alter to multiply correctly across each row, giving you a sale price from the cost entered in column C.

This is a fairly simple example of the calculatory powers of an ST spreadsheet. To include VAT mark-up as well, you only need to add an extra column – say in column E, forcing Price into column F – of VAT mark-up of 1.15. Then a formula of C1*D1*E1 would give the extended calculation to give a new sale price in column F.

An expenses ledger

A SIMPLE expenses ledger like the one shown in Figure III can be set up by using some of the principles already discussed. Using Graphic Sheet you can construct a ledger template. The column and row headings can be entered in a similar fashion to the way we entered them for the price list. The only difference being with Graphic Sheet you must press Return after each entry.

Then the relevant expense figures for each category in each month can be entered as and when desired. Referencing a range of cells, your sheet can calculate total monthly expenses at the end of each column and total category expenses at the end of each row.

This is done by setting a formula for a range of cells. For instance, the sum of all cells included in January is delimited by the range R3C3 (row 3, column 3) to R9C3 (row 9, column 3). With Graphic Sheet it can be written like this =SUM(R3C3:R9C3), the character : indicating a range. The formula will successfully be applied to January and will give a total for monthly expenses in the desired cell, in this case cell R11,R3.

This same formula can be dragged with your mouse on to the other 11 months and copied to calculate the totals for those months. A similar formula can be constructed for the rows to show category totals at the end of each row.

Keeping track of your bank account

IT is also possible quickly to produce a sheet to keep an eye on your current or business bank account. The spread in Figure IV gives an indication of the type of simple check that can be made. Enter the heading details as before, in the same manner as they are produced on your bank statement.

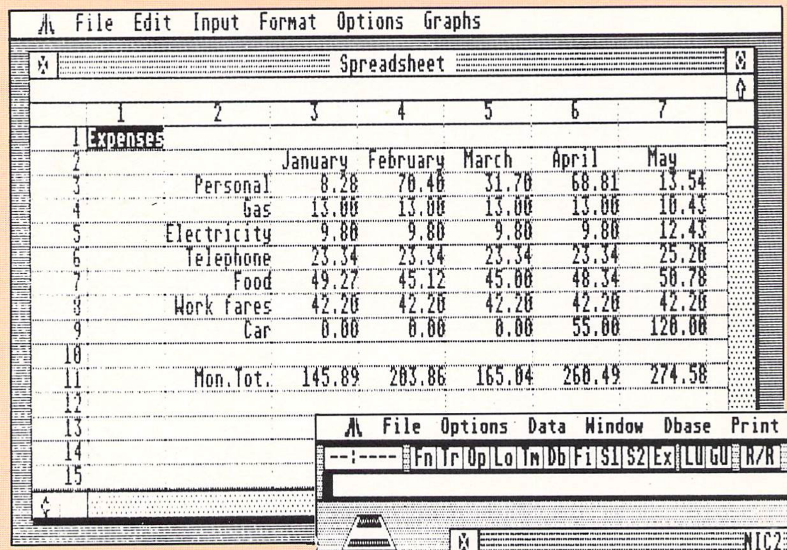
Now you must predefine the date display in column A. Like most other ST spreadsheets, K-Spread 3 allows you to set up the sheet in advance in readiness for data to be added later. The date may be primed as a label by selecting Preformat from the Data menu and by clicking on the create button. Follow a simple on-screen procedure and any data entered into column A will be converted automatically into date format.

Column B can be preformatted in much the same way – this time as labels. Columns C and D – the credit and debit columns – will be numbers, therefore no previous formatting is necessary as spreadsheets automatically recognise numbers.

In column E, row 1 will be the balance brought forward. From Row 2 onwards we can set up a formula to automatically calculate the balance when a new credit or debit is entered.

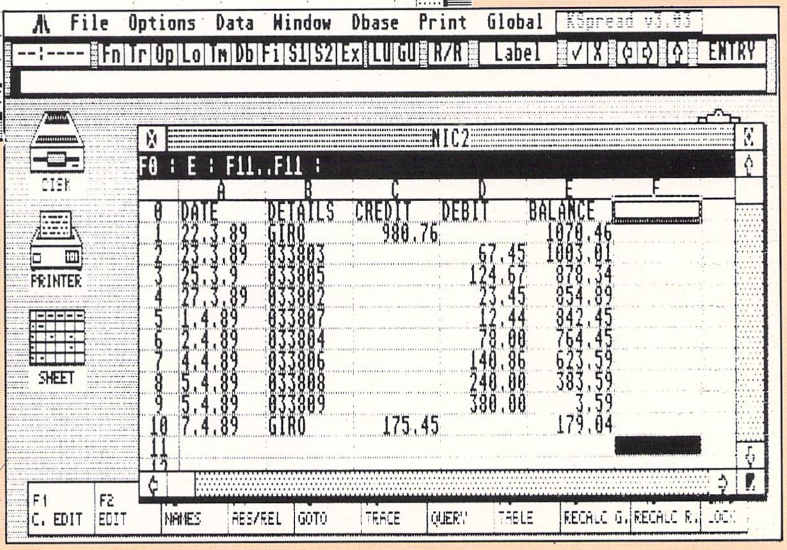
In this case the formula would be IF(C2-D2<>0 THEN E1+(C2-D2)ELSE'' ''). To avoid entering the formula many times, it can be copied down column E by dragging cell E2 to the column header and setting the range as before. Now you have a powerful bank statement manager which can be updated automatically at any time as credits and debits arise.

The above examples are just the tip of the iceberg, but will give you the inside knowledge to begin making practical use of your spreadsheet for important business purposes.



	1	2	3	4	5	6	7
1	Expenses						
2			January	February	March	April	May
3		Personal	8.28	70.40	31.70	68.81	13.54
4		Gas	13.00	13.00	13.00	13.00	10.43
5		Electricity	9.80	9.80	9.80	9.80	12.43
6		Telephone	23.34	23.34	23.34	23.34	25.20
7		Food	49.27	45.12	45.00	48.34	50.78
8		Work fares	42.20	42.20	42.20	42.20	42.20
9		Car	0.00	0.00	0.00	55.00	120.00
10							
11		Mon.Tot.	145.89	203.86	165.04	260.49	274.58
12							
13							
14							
15							

Figure III: An expenses ledger created using Glentop's Graphic Sheet



F0 : E : F11..F11 :

	A	B	C	D	E	F
0	DATE	DETAILS	CREDIT	DEBIT	BALANCE	
1	22.3.89	GIRO	980.76		1070.46	
2	23.3.89	033803		67.45	1003.01	
3	25.3.9	033805		124.67	878.34	
4	27.3.89	033802		23.45	854.89	
5	1.4.89	033807		12.44	842.45	
6	2.4.89	033804		78.00	764.45	
7	4.4.89	033806		140.86	623.59	
8	5.4.89	033808		240.00	383.59	
9	5.4.89	033809		380.00	3.59	
10	7.4.89	GIRO	175.45		179.04	
11						

Figure IV: A flexible bank statement which can be automatically updated

Getting things sorted out

ANY business worth its salt will need to keep track of its suppliers and customers. This can often be a laborious process of making and updating hand-written mailing lists and directories of important phone numbers. However, all this can be simplified and speeded with the aid of your ST and a database program.

A database or datacard program will allow you to create a filing system on disc of important information which can then be quickly and easily accessed, updated, searched and sorted as required. Total or selected information can then be output at will to your printer.

A typical public domain database program will allow up to 4,000 quite detailed records. Even this hefty number can be increased by sub-indexing on a number of separate discs. For example six discs can contain 4,000 records each – totalling 24,000 records in all.

The commercial equivalents have even more breathtaking capabilities, allowing calculatory graphics effects such as graphs and pie charts, and data storage of over 40,000 records in one file. And some of the more recent professional database packages – for example, Superbase Personal – allow single files to contain up to 999 indexed fields (specified lines of information).

Add to this the fact that individual text fields can have a limit of more than 250 characters – even greater with some software when importing data from external packages – and you get some measure of the power of a top notch ST database program.

Databases

THE joy of an ST-run database program is that you can store your own personal database filled with any information you like on one small 3.5in disc.

Such programs are ideal for storing customer and supplier names, addresses and telephone numbers as well as intrinsic business details relevant to each business client.

Figure I shows a typical page from a simple datacard – a smaller and less powerful brother of the fully fledged database – program. Of course, exact record details can be briefer or much more detailed than shown.

Once you have compiled a database of records you can search quickly for any selected piece of information. For example, imagine you wish to construct a list of all suppliers of a particular product in the county of Sussex. This can be achieved by operating a simple Search command, for the string "Sussex".

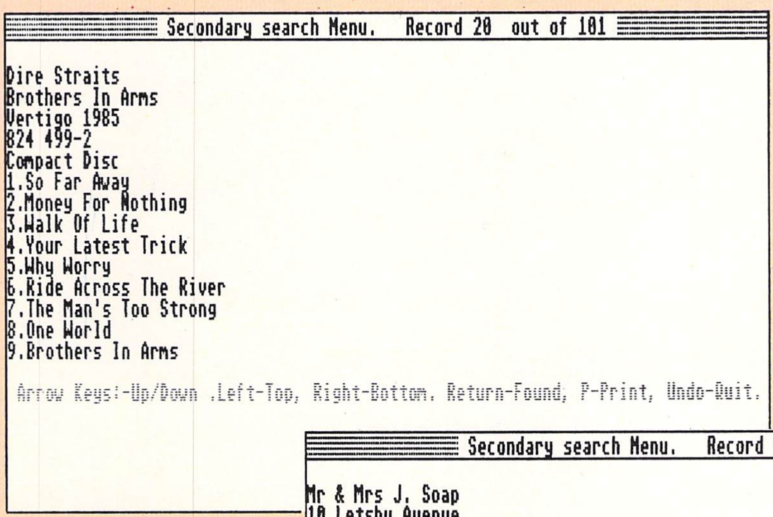
Equally you can sort your data into either numeric, date or alphabetic order by using a standard Sort command. Most commercial database programs also supplement powerful search and sort facilities by a process known universally as mail merge.

This allows you to merge data you have collected into a compatible word processor to produce address labels, macro letters, forms and other duplicated mail that needs a personal touch. This saves time which would have been formerly spent on repetitious duplication of names and addresses.

Such merge compatibility exists between several products, such as Kuma's K-Data with K-Word 2, and Timeworks Data Manager with Word Writer ST.

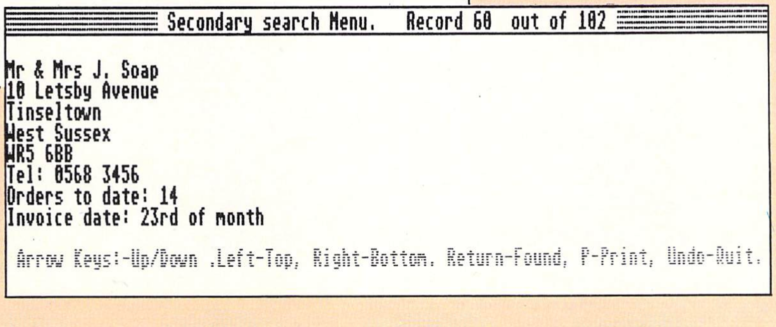
Simple database and index card programs are also freely available. Some allow field size – length of each line of information – to as many as 3,000 characters.

It is worth scanning PD Software library catalogues for some existing samples such as Database I. In addition, programs such as B.Bytes' B Base 2, shown in Figure II, costs less than £15 and provides good cataloguing facilities and room for up to 7,500 names and addresses on a double sided disc.



```
Secondary search Menu.   Record 20  out of 101

Dire Straits
Brothers In Arms
Vertigo 1985
824 499-2
Compact Disc
1.So Far Away
2.Money For Nothing
3.Walk Of Life
4.Your Latest Trick
5.Why Worry
6.Ride Across The River
7.The Man's Too Strong
8.One World
9.Brothers In Arms

Arrow Keys:-Up/Down .Left-Top, Right-Bottom. Return-Found, P-Print, Undo-Quit.
```

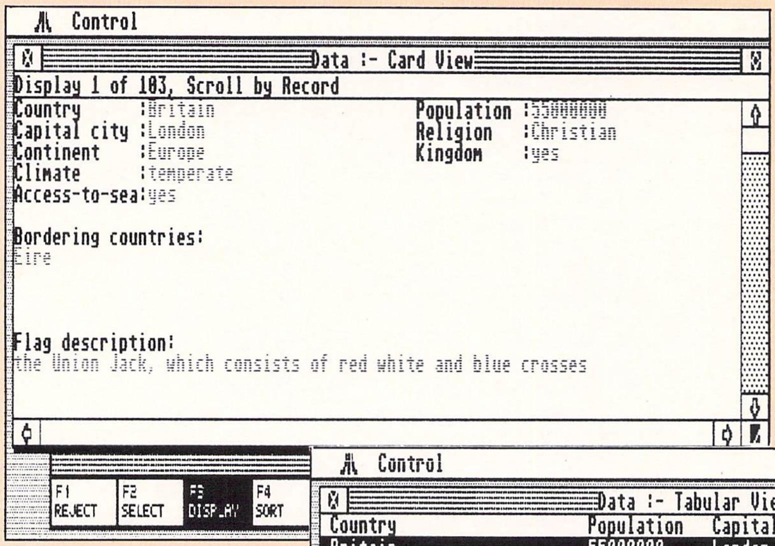
Figure I: A typical page from a datacard program



```
Secondary search Menu.   Record 60  out of 102

Mr & Mrs J. Soap
10 Letsby Avenue
Tinseltown
West Sussex
WR5 6BB
Tel: 0568 3456
Orders to date: 14
Invoice date: 23rd of month

Arrow Keys:-Up/Down .Left-Top, Right-Bottom. Return-Found, P-Print, Undo-Quit.
```

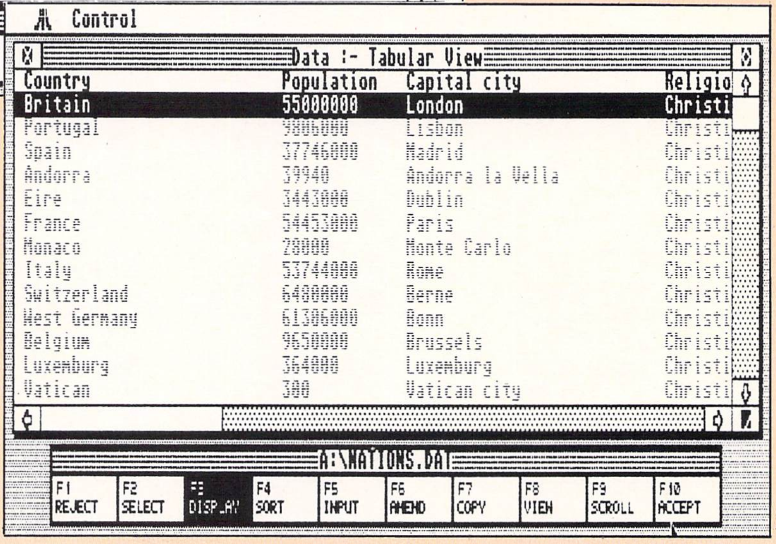
Figure II: A completed single record using B Base 2

Getting things sorted out



```
Control
Data :- Card View
Display 1 of 103, Scroll by Record
Country       :Britain               Population :55000000
Capital city  :London                Religion   :Christian
Continent     :Europe                Kingdom    :yes
Climate       :temperate
Access-to-sea:yes

Bordering countries:
Eire


Flag description:
the Union Jack, which consists of red white and blue crosses
```

Figure III: A completed record using Data Manager Professional



```
Control
Data :- Tabular View
Country            Population  Capital city        Religio
Britain            55000000    London              Christi
Portugal           9806000     Lisbon              Christi
Spain              37746000    Madrid              Christi
Andorra            39940       Andorra la Vella    Christi
Eire               3443000     Dublin              Christi
France             54453000    Paris               Christi
Monaco             28000       Monte Carlo         Christi
Italy              53744000    Rome                Christi
Switzerland        6480000     Berne               Christi
West Germany       61306000    Bonn                Christi
Belgium            9650000     Brussels            Christi
Luxemburg          364000      Luxemburg           Christi
Vatican            300         Vatican city        Christi
A:\NATIONS.DAT
```

Figure IV: An index of data files

Using a database

Filling in a database with the information you require is a simple and speedy task once you become familiar with a few procedures. After you decide what type of information you want to store you must construct a format to hold that data in an ordered and clear manner.

Imagine you want to create a name and address list of business clients. Think of the data format as a form that you fill in for each of the people on your address list. This form would have a line for each piece of information about one person on the list, and each line would have a title so you would know what to write on that line as in the following example:

First name:	....
Last name:	....
Address:	....
Town:	....
County:	....
Post code:	....
Phone:	....
Business name:	....
Product:	....

Each of these lines is called a field. The data format shown above consists of nine fields. With most database programs you can specify the type of information you want to store in each field. This can be either numeric – numbers only – or textual. Some databases also allow fields to contain alphabetic data – single letters only – or calculatory form.

Due to the nature of the way the program sorts, information sorting will be quicker and easier on alphabetic or numeric fields.

Not only will you be able to choose what type of information will be stored in each field but you can also designate how long each field will be. However, due to memory constraints it is usually true that if you increase the length of each field you decrease the number of records the database can hold.

When you fill out your format for a single person on your list you create one record. Each person's information is entered into a blank copy of the format, much like filling out a separate card for each one and keeping it in a card index file. An example of a completed database record using Data Manager Professional can be seen in Figure III.

The only difference is that when you finish filling in one record it will be stored in the ST's memory and can then be stored on your data disc.

Each record is numbered by the program in consecutive order. A group of records is called a data file. An index of records on part of a data file is shown in Figure IV.

Product	Supplier	Price
B Base 2	B.Bytes	£14.95
Mini Office Professional	Database	£24.95
Data Manager	Timeworks	£39.95
ST Organiser	Triangle	£49.95
K-Data	Kuma	£49.95
Data Manager Professional	Timeworks	£69.95
H&D Base	Silica	£99.95
Laserbase ST	Laser	£99.95
Superbase Personal 2	Precision	£99.95
dBase II	First Software	£119.00
BAS Database	BAS	£138.00
dBMan	Atari	£194.00
Superbase Professional	Precision	£249.95

Some common database packages for the ST

WE have seen the range of database packages available for the ST and looked at examples of their flexibility and speed. The strength of an ST run database is that you can store your own personal filing system filled with any information you like on one small 3.5in disc.

A database – database manager to use its correct terminology – can save a business important time and money in such previously laborious tasks of collating and updating mailing lists and directories of important phone numbers.

However, like any business software package, a database manager can be difficult for a beginner to come to terms with, in its manipulation and overall working.

Here we look at how to start getting your database to work for you.

A database tutorial

LET'S conduct a simple tutorial to give an insight into the direct workings of a database manager program.

For simplicity we will use the ever popular and industry standard Data Manager by Timeworks. Though other packages may differ slightly in some respects, most of the terms and methods used are universal.

You will need, not only the program disc, but also a blank formatted disc for your records (data).

● Load the program disc in the usual way.

● When the program is loaded click on the Cancel box, as we will be creating a new data file.

● Move the cursor to the Options menu and select the Create New Column menu.

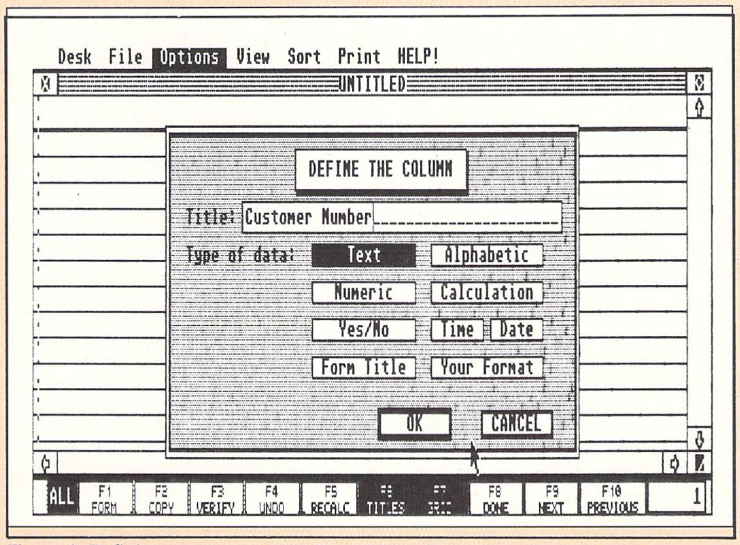
● Enter the title *Customer Number* for your first field in the dialogue box – seen in Figure I.

● Now click on the Text box and this will set the type of data we wish to store. Click on OK.

● Create the second field by selecting Create New Column again. Enter *First Name* and click on the box marked Alphabetic to select the type of information – as you may wish to sort this information into alphabetical order.

● Now enter your third field in the same manner and enter *Last name* as the field title. Once again choose Alphabetic as the type of data to store.

● Enter four more fields in a similar manner and title these: *Address, Town, County, Post Code*. Select Text as the type of information in all cases – even



Title: Customer Number

Type of data: Text

Figure I: Defining a column using Data Manager

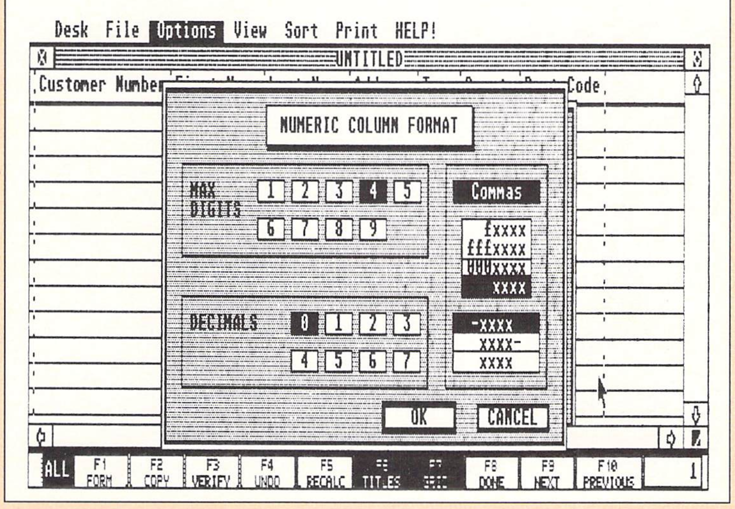
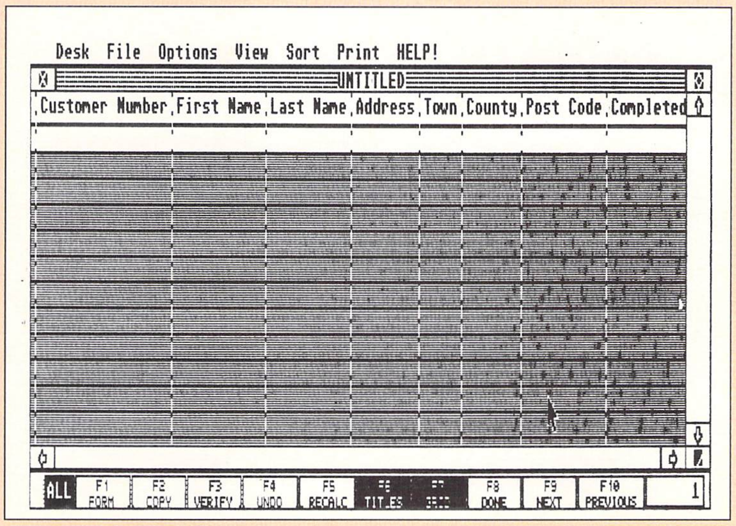


Figure II: The numeric dialogue box



Customer Number, First Name, Last Name, Address, Town, County, Post Code, Completed

Figure III: The field titles as saved

Electronic filing

though post codes include numbers as well as text.

● We will now enter the last field name as *Completed Orders*. As this is numeric it will require one more step than the others.

A dialogue box will appear – as shown in Figure II – which will ask you to specify a number of digits – select 4 – for the field and a number of decimal places at the right of the decimal point.

We do not need any decimal places with completed orders, so click on the 0 box before clicking on OK.

● Your file should now look like the screen shown in Figure III.

● It is a good idea to save the file format on your blank data disc. Do this by placing your formatted disc in the drive and select the Save As option from the File menu.

● When the dialogue box appears, enter the name *TUTORIAL.DMF* – Tutorial Demonstration File – for your new database and press Return. Your format will be saved on disc.

● Now you can begin inputting data, thus creating records for your data file. To begin adding records move into what is called the Form Style display by selecting Form Style option on the View menu.

● You will find the cursor is placed after the title of the first field. Enter 1001 and press Return. If you make a mistake, use the cursor keys to move through the field and type over the error.

● Now enter the name Julia in the *First Name* field and press Return. Then continue entering information in this record using the example input below. Remember to press Return after each entry:

Last Name: Smith
Address: 24 West Street, Blit Village
Town: Megatown
County: Middlesex
Post Code: ME4 6AT
Completed Orders: 15

● Your completed record should now look like that displayed in Figure IV, again it is a good idea to save to disc at this stage.

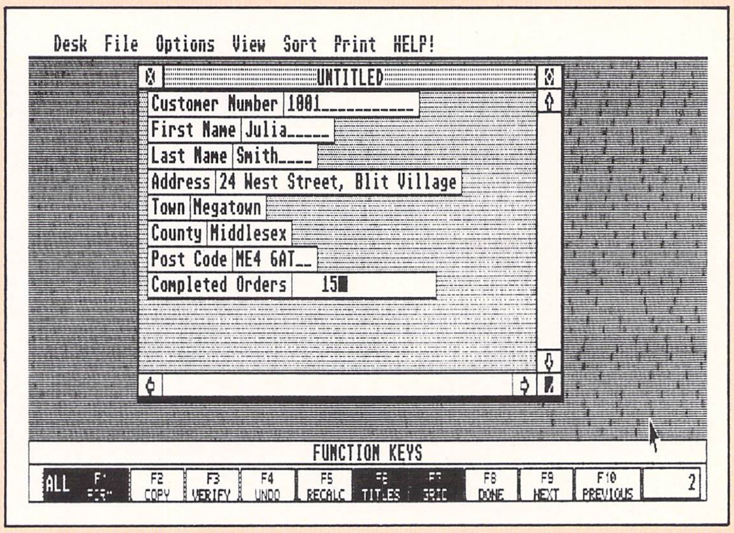
● If an entry is not entirely visible in a particular field, use the mouse to drag the field to the right, thereby enlarging your information entry area.

● Enter a few more records in a similar manner, using any information you like. Then select the Save option on the File menu to store your database format and records on your data disc.

● Now experiment with your sample data and the options in the main program. Try deleting a record, modifying a record or performing a search. Don't worry, you can't harm the sample data.

Hopefully this will get you started and help you learn a little about the inbuilt processes of a database manager.

In turn it will give you confidence to use some of the more powerful facets of databases such as calculatory, merging and graphics routines.



Customer Number 1001__________
First Name Julia_____
Last Name Smith____
Address 24 West Street, Blit Village
Town Megatown
County Middlesex
Post Code ME4 6AT__
Completed Orders 15

Figure IV: A completed record

GLOSSARY

Backup	A duplicate set of data to be used in case the original is lost, destroyed, or accidentally altered.
Character	A single letter, number, symbol or space.
Chronological	Information sorted by date.
Classes	Created by extracting certain information from the main datafile by applying a rule or definition. The rule acts as a filter, extracting only the information which meets the specification. Sometimes known as a sub-datafile.
Data	The information you enter to be processed.
Database format	The arrangement of data within a record. The format can be designed to hold data in labelled information entry lines (fields).
Datafile	A complete database which has been saved on disc. Sometimes simply called a file.
Dataset	A group of information designated for drawing a graph. Each data set includes the numbers and labels that will appear on the graph.
Dialogue box	An on-screen box to enter information or choose from options.
Enumerated	Pre-defined values within fields.
Field	An entry line of information.
File	A database which has been saved on a data disc.
Match	A record found by the program in a Search. The program matches the data that you have requested to the data in the record.
Merge	Allows you to enter existing data into a datafile, add records in one datafile to another, or divide records in a datafile into two by creating two classes. Also applies to entering existing data into a word processor program to create macro letters and multiple pro-formas.
Record	One or more fields linked together like a chain to form a single specific piece of information.
Report	Output of data of your choice to screen, disc or printer. Reports may be saved, loaded and edited from disc for future use.
Search	A systematic examination for specified information.
Sort	To arrange items of information in a desired sequence.
Verification	A process by which the program looks to see if information has been repeated in more than one record.

First steps in Basic programming

IN this section we'll be demonstrating how to program in ST Basic, from raw beginnings up to some of the more complex features. If you've no idea where to start, this series is for you

First principles

ESSENTIAL to successful programming of any sort is to realise that computers don't think. Surprising as it may seem, today's computers cannot reach logical conclusions about anything on their own. They give the impression of rational thought by their actions, while all they are doing is following a sequence of instructions – a program.

Consider a simple everyday task like making a cup of tea. Everyone can cope with this without even thinking about it. It's the sort of thing we'd expect a six-year-old to manage. But just how is it done? In fact we are executing a very complex sequence of events – a program if you like.

Try this...

NOTE down exactly how you would make a cup of tea, in the form of a list.

When you've finished, take a look at Figure I overleaf. If your list looks something like it, don't be too surprised – it contains what most people would probably consider to be the important features in tea making. When it comes to the computer though, it doesn't have the first clue about any of this and has to be led by the hand.

Now consider the revised example in Figure II. Ask yourself: Is it simple or complex? It looks incredibly complex, although it is in fact the first example broken down into extra stages. The three items shown in brackets would normally be broken down still further.

Using Figure II, try expanding the sections in brackets. How many steps can you divide the process into?

Getting started

ASSUMING you haven't already done so – make a copy of the UK language disc and keep the original in a safe place. You'll find details of how to make a copy in the owner's manual supplied with your ST. Next insert the copy into drive A and press reset.

When the desktop appears, double click on the BASIC.PRG icon – after a few seconds the ST Basic desktop will appear. This is made up of four windows: Command, List, Output and Edit with the Edit window hidden under the other three. The functions of these windows are:

● **Command:** The main window. This is where you will type either direct instructions to Basic or the lines of your program.

● **Output:** Basic will display its results here – either from a direct command or from a program.

● **List:** This is where the listing of any program you write will appear line by line.

● **Edit:** When you make mistakes in your program lines – and you will – this is where you edit them. It is possible to re-type the offending line but usually quicker to edit it.

These four windows are, therefore, Basic's interface with you. Through them you can talk to Basic and Basic can talk to you: You give instructions and receive results.

What is Basic?

SINCE the very early days of microcomputers the word Basic – an acronym for Beginners All-purpose Symbolic Instruction Code – has become part of our everyday computerspeak. Although it's a very simple language for newcomers to get to grips with, it is still a very powerful programming tool – despite the derogatory noises made about it by computer literates.

PRINT

TO show how the two main windows work, first ensure the Command window is active. If it isn't just click the mouse anywhere inside its boundary. Now type:

```
PRINT "Hello world!"
```

and press Return. The message in quotes will appear in the Output window. If you made a mistake don't worry, just type the line again.

Direct commands

WHAT you have done in fact is given Basic a direct command to write the message *Hello world* to the screen. Direct commands are at the simplest level of program control and the PRINT statement is one of the simplest commands available in Basic. It's used whenever a program has to display information or results. In fact it's used so often that ST Basic provides an abbreviation – the ? (query). The two are interchangeable but you would be wise to stick to one or the other.

The words following PRINT are what will appear on the screen. They have to been enclosed in quotes because we want them displayed literally. If the quotes are omitted Basic expects to find a variable – more of those later.

The Basic differences

WHEN Atari first released the ST range, the Basic which accompanied the machines was not quite as refined as the one we know today. Superficially they seem the same, but the original version contained a number of bugs which at best were annoying and at worst could crash the system.

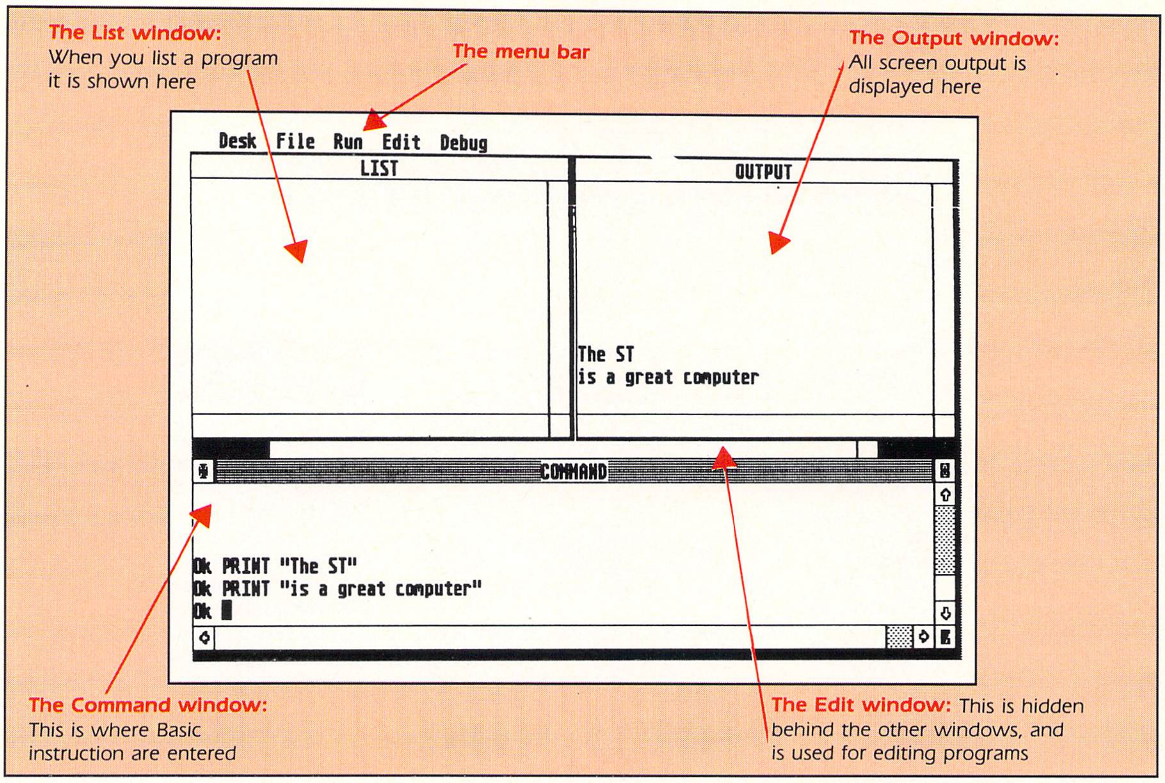
These have now been ironed out, and if you bought your machine since October last year you should already have the updated version. For those who aren't sure there is a simple way to check which revision you have. After loading Basic pull down the Desk menu and click on *About ST BASIC*. If the copyright date shows 1987 you have the new version.

In the meantime, the more ambitious among you should watch out for programs which include PEEKs and POKEs, or the SYSTAB, GEMSYS and VDISYS arrays, as these are the most likely areas to cause problems. We'll be going into more detail about the differences much later in this section when they may start to affect the examples on these pages.

If you've still got an old version and wish to update, contact one of the specialist ST dealers and ask them if they can put it on a disc for you.

The full manual is available separately at £9.95, which should also please all those who received only the small reference pamphlet with their new machine.

First steps in Basic programming


The List window: When you list a program it is shown here
The menu bar
The Output window: All screen output is displayed here
Desk File Run Edit Debug
LIST
OUTPUT
COMMAND
The Command window: This is where Basic instruction are entered
The Edit window: This is hidden behind the other windows, and is used for editing programs

```
The ST
is a great computer
```

```
Ok PRINT "The ST"
Ok PRINT "is a great computer"
Ok
```

The ST Basic desktop

```
1. Fill kettle
2. Boil kettle
3. Put teabag in pot
4. Pour on boiling water
5. Pour some milk into cup
6. Pour in tea
```

Figure I: Steps in making a cup of tea

```
 1. Grasp kettle by handle
 2. Take kettle to sink
 3. Grasp and remove lid
 4. Place kettle under cold tap
 5. Turn tap two turns anticlockwise
 6. Wait until kettle full
 7. Turn tap two turns clockwise
 8. Replace lid, free grasp
 9. Plug in kettle
10. Switch on kettle
11. Wait until water boils
12. Switch off kettle
13. Remove teapot lid
14. (Insert tea bag)
15. Pour in boiled water
16. Replace teapot lid
17. (Put milk in cup)
18. (Pour tea)
```

Figure II: The tea making steps broken down still further

Try this...

TYPE the following, pressing Return after each entry:

```
PRINT "The ST"
PRINT "is a great computer"
```

and Basic responds:

```
The ST
is a great computer
```

Notice that Basic responds by printing *This is* first then prints the rest of the text on a separate line. This is because when Basic reaches the closing quote of a print statement, it also generates a new line. You can suppress this using a semi-colon (;) after the closing quote as in:

```
PRINT "The ST ";
PRINT "is a great computer"
```

which produces:

```
The ST is a great computer
```

Notice the extra space before the closing quote in the first statement. Basic only prints what it finds: Omitting the final space would not produce the correct result. But don't take my word for it. Try it and see for yourself

Basically...

PROGRAMMING of any sort is all about solving a problem – by breaking it down into manageable chunks the computer can understand. This is true of all programming languages – not just Basic.

...and this

THE comma also has its uses. When Basic prints a line it divides it into zones or fields each 14 characters wide. These are similar to the tab stops on a typewriter or word processor. By using the comma the invisible print cursor can be moved to the next available tab stop. To see this in action type:

```
PRINT "Zone 1","Zone 2"
```

which prints:

```
Zone 1        Zone 2
```

Notice that the two items are separated by a comma but still form part of the same print statement. Notice also, that the second item is printed 14 characters from the border – not the last piece of text.

Exercises

● PRINT is normally followed by a set of quotes enclosing some text. What happens if it is used on its own?
● A semi-colon will suppress linefeeds after print statements. What happens if it is placed before the opening quote?
● A comma sends the print cursor to the next tab stop. What happens if you place more than one comma together?
● Experiment using the comma and semi-colon directives in different combinations. Try forecasting the results before pressing Return.

Basically...

WHERE computers score a big hit is in their speed. They execute (carry out) the instructions given to them extremely quickly, so even if we give the machine a lot to do, it'll still be done at lightning speed.

Our first program

Simple arithmetic

DOING maths in ST Basic is essentially no different from doing maths on a pocket calculator. However, the symbols used for the various mathematical functions frequently bear no resemblance to their algebraic counterparts, which often results in confusion. You can see in the panel below the most common symbols and their meaning.

At this point you may well be wondering where the equals (=) sign has gone. In fact, equals is used frequently in Basic, but isn't necessary when we require an immediate answer.

Let's try some simple addition sums. Enter the following, not forgetting to press Return at the end of each line:

```
PRINT 2+3
PRINT 2+3*2
```

This last simple sum has two obvious answers 8 or 10. Basic, correctly prints 8. Why? See the panel below for the answer.

Variables

IF Basic could only work with absolute values – 2, -5, 4.5 are examples – programs would be either incredibly long or impossible to write in the first place. To avoid this, Basic enables you to use variables. These are similar to the letters we use in algebra, but don't worry if this topic makes your brain itch – it's much easier in Basic.

You can use almost any combination of letters and numbers to make up a variable's name. However, the first character must not be a number and only the first eight are unique.

Basic has four types of variable: Single and double precision reals, integer and string. For the moment though we'll stick with the simplest – single precision real. At this stage, although they sound quite a mouthful they're really quite easy to understand.

Basically...

A PROGRAM is far more readable if variables are given meaningful names. For instance, in a program to keep track of your bank balance you could call the final amount *balance*. You can join words together with an underscore (new ST Basic) or a full stop (old ST Basic) like *bank_balance*, and *bank.balance*.

What's in a name?

All variables have names, and it's good practice – although not essential – to give them names that mean something. This way when you try to modify a program months later, you'll have some chance of understanding it.

Let's take a simple example: John has five apples and six pears, and he wants to know how much fruit he has. In terms of variables the problem looks like this:

```
pears = 6
apples = 5
fruit = apples + pears
PRINT fruit
```

or in a shorter, less clear form:

```
p = 6
a = 5
f = a + p
PRINT f
```

From this basic premise, it's easy to see how variables can be set or altered to suit the requirements of the program. The equals symbol is used here as an assignment operator – that is, a variable is assigned (given) a value.

Oh, my giddy Aunt

At school we are taught to evaluate expressions using the My Dear Aunt Sally (MDAS) rule of thumb. That is, multiplication first, followed by division, then addition and lastly subtraction. Not surprisingly, Basic computes its sums in exactly the same way.

This system is often referred to as operator precedence, but it means the same thing. In the example above therefore, Basic computes the multiplication first – then the addition.

Of course, there are cases when you need to force Basic to evaluate an expression in a particular order. For instance, to do the addition first. This is done by surrounding such parts in brackets. If the result of the previous example was intended to be 10 then the addition – which has a lower precedence than multiplication – has to be enclosed in brackets. Now try this:

```
PRINT (2+3)*2
```

which prints 10 as you might expect.

Brackets can, if necessary be nested to give some gloriously complex equations.

For example:

```
PRINT 1+(((2+3)*4)+5)^6
```

In practice it's very rare to see an equation as complex as this. For one thing it's almost impossible to read, and for another it's very prone to errors – for instance, not enough closing brackets, or brackets in the wrong place.

Before leaving pure arithmetic, there are two operators requiring further investigation, exponentiation and modulus.

The table below shows these as symbols, although you should note that modulus is a keyword (MOD) not a symbol, not that it makes any difference to Basic.

I'll leave you to discover what they do for yourself.

A ^ B	Raise A to the Bth power (exponentiation)
A * B	Multiply A by B
A / B	Divide A by B
A MOD B	Divide A by B and leave the remainder
A + B	Add A to B
A - B	Subtract B from A

Arithmetic symbols in ST Basic in order of precedence

Our first Basic program

I have left this until now because it introduces a whole new part of ST Basic – the List and Edit windows. Everything we have done so far can be typed directly into the Command window, and the result of our instructions is echoed in the Output window. Entering a program is a little more complex, but mistakes can be rectified relatively easily once you get the hang of the editor.

Type EDIT from the Command window to enter the editor, and F10 at any time to leave it. While a line is being edited it will be shown in light text, but note that the changes don't take effect unless you press Return. To get used to the editor it may be worthwhile to use it to enter the following examples.

All programs in ST Basic rely on line numbers as reference points. Every instruction you give to the computer is typed on a line, and every line has a number. It is a long standing convention that line numbers go up in steps of 10 – 10, 20, 30 and so on. This makes it easier to insert extra lines at a later stage.

Incidentally, the Basic command AUTO can be used to enter the line

Our first program

Try this...

- Experiment with different expressions using *, /, +, and - Try to predict the results?
- Variables are normally assigned a value. What happens if you attempt to use a variable before it has been given a value?
- Modify the fruit program to account for the difference in price between apples and pears.

numbers for you, though I prefer not to use it.

Enter the following short program using the editor:

```
10 INPUT "How many pears ",pears
20 INPUT "How many apples
   ",apples
30 fruit = pears + apples
40 PRINT "Total = ";fruit
```

The first two lines introduce another simple keyword, INPUT. This causes the program to stop and wait for you to type something and press Return.

In this case the program will expect to find a number. Anything between the quotes is printed just like the PRINT statement. This time though, when you enter a number and press Return, the value is assigned to the variable following the comma.

The message in quotes is in fact optional, and if you omit it Basic responds by stopping and printing a question mark while it waits for some response.

To see this program work, type RUN or operate RUN from the menu of the same name. Of course, a program like this can be simplified. For instance line 40 can be rewritten as:

```
40 PRINT "Total = ";
apples+pears
```

Which completely obviates the need for line 30.

Now let's just suppose John buys the fruit at a fixed cost per unit, no matter what it is. How does he calculate how much he has to pay? We need another variable *cost* to indicate the cost of the produce.

This amount will be fixed, so it can be defined as a constant in the program. In Basic constants are just variables – it's up to the programmer to ensure they keep the same value. Here's our program modified to include the cost per item:

```
 5 cost=0.10
10 INPUT "How many pears ",pears
20 INPUT "How many apples
   ",apples
30 PRINT "Total="apples+pears
40 PRINT "Price="(apples
   +pears)*cost
```

Variable differences

THERE are quite significant differences you should be aware of in the way old and new ST Basic handle variables. For instance, old ST Basic stores integer variables – those with a per cent sign tagged on to the end of their name – in two bytes. This means they can handle whole numbers between -32,768 and +32,767.

New ST Basic on the other hand uses four-byte integers with a much larger range of -2,147,483,648 to +2,147,483,647. Similarly, double precision floating point numbers – those with a hash after their name – can be much larger in new ST Basic.

There are also bugs in old ST Basic. For instance, try entering:

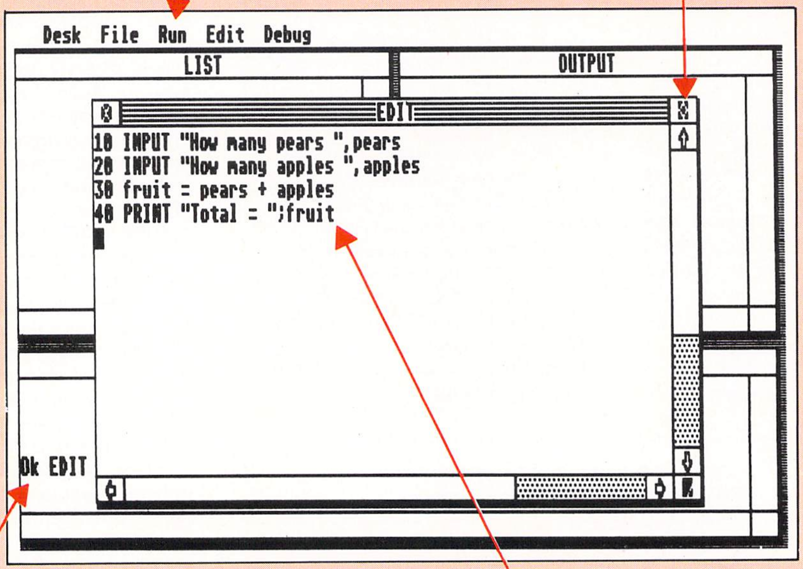
```
x = 77777
```

and you'll be told *function not yet done*, and if you print out the value of *x* you'll find it is 0.00079E+11! So take care with floating point maths operations – the results may not be what you expect.

Basically...

STRUCTURED programming involves breaking a problem down into small meanageable chunks or modules. These modules can be written as general procedures and are nromally complete routines which are totally independent of the rest of the program.

Once written they can easily be incorporated into any program. Each module will have a specific function, and possibly, input and output parameters. A whole team of programmers may work on large projects, and a module may be allocated to each one.



```
10 INPUT "How many pears ",pears
20 INPUT "How many apples ",apples
30 fruit = pears + apples
40 PRINT "Total = ";fruit
```

Pull-down this menu with the mouse and click on Run to see the program in action

Click here with the mouse to make the Edit window full size

Type EDIT in the Command window to enter edit mode, or pull down the Edit menu and click on Edit

The Edit window is brought to the top and you can enter your program here

Real programming

Real numbers and integer variables

THE single precision real number is perhaps the simplest of all variables in ST Basic because it behaves exactly as you would expect. Real numbers are any numbers that can contain a floating point fractional part – a decimal fraction.

However, if the thought of decimal fractions makes you twitch don't worry – they're just like amounts of money. For instance, the number 3.14 is an approximation of the mathematical constant PI, but it could equally mean £3.14 – the price of four pints at the local.

Similarly the numbers 1.2345679 and 1.0 are both real numbers. They have an integer (whole) part and a fractional part. However, 1 is an integer and doesn't have a fractional part.

In fact computers can only deal with integers – binary zeros and ones – the conversion to real numbers being performed at a very low level by clever software.

It makes sense then, that this conversion process takes time, and if an alternative was provided programs would execute faster.

The advanced features of ST Basic provide for this in the form of so-called integer variables – variables that can only hold whole numbers. At first sight these may seem a little limiting, however, most programmers rarely use anything else, unless floating point arithmetic is specifically called for.

Even then, many use techniques known as scaling and rational approximations. The table here shows some rational approximations for some typical mathematical functions and numbers.

Number	Approximation	Error
PI = 3.14	355/113	8.5 x 10E-8
SQR(2) = 1.414	19601/13860	1.5 x 10E-9
SQR(3) = 1.732	18817/10864	1.1 x 10E-9
e = 2.718	28667/10546	5.5 x 10E-9
c = 2.99792	24559/8192	1.6 x 10E-9

Specifying a type

THERE are two ways to describe an integer variable in ST Basic. The first is to append the per cent character to the end of its name, and the second is to use the DEFINT statement. Per cent can be used after any variable name to denote that it is an integer, and it works in immediate mode or within a program listing.

DEFINT can only be used inside a program, but works on a range of variables. For instance the line:

```
10 DEFINT a-d
```

tells Basic that any variable name starting with a letter between *a* and *d* inclusive is to be treated as an integer. You should note this statement is not case sensitive, which means variables starting with *A-D* are also treated as integers.

Incidentally, the exclamation mark type specifier can be used like the per cent sign to indicate a real number, overriding the DEFINT for the named variable.

Integer arithmetic is the simplest form of all maths, but you should be wary when attempting to mix integers with real numbers. Try the following in immediate mode – type it into the command window – being careful to note the use of per cent:

```
A%=2
B=2
PRINT A%+B
```

Basic prints 4, which is correct. Now try the following example:

```
A%=2.5
B=2.5
PRINT A%+B
```

This prints 5.5, which is quite wrong. Why?

What happened is that Basic has performed rounding on the integer variable *A%* and made it 3. This operation is performed whenever the variable is assigned a new value – so the error can be cumulative and very confusing. Remember, if you must mix types – do so with extreme caution.

Just to confuse matters further, Basic provides the INT statement to slice off the fractional part of a real number. Try the following to see this in action.

```
A=2.9
B%=A
PRINT "Value=";A
PRINT "Rounded=";B%
PRINT "Truncated=";INT(A)
```

This prints:

```
Value=2.9
Rounded=3
Truncated=2
```

A function related to INT is CINT. It works in the same manner to INT but this time it rounds the number instead, so the previous example could be written:

```
A=2.9
PRINT "Value=";A
PRINT "Rounded=";CINT(A)
PRINT "Truncated=";INT(A)
```

Strings and things

IF by this point you're still eagerly typing in the examples then you've got what it takes to be a programmer. If on the other hand, all of the maths is starting to make your brain wave the white flag, don't worry. Programming fortunately, is not all about complex maths.

The third variable type in ST Basic is called a string, and these have the dollar character appended to their names. For instance, *A$* and *name$* are string variables. They are used to store sequences – strings – of characters, letters, numbers and so on.

Unlike numeric variables, you can only add strings together, a process called concatenation. Attempting anything else, like division, would be meaningless to Basic, and causes an error.

At first sight then, strings may seem a little limited. Not so. They are one of the most powerful features of the Basic language. Try out the following program:

```
10 INPUT "Last name:";last$
20 INPUT "First name:";first$
30 PRINT "Hello ";first$;"";last$
```

The computer responds:

```
First name:?Mark
Last name:?Smiddy
Hello Mark Smiddy
```

Now by juggling the program around a little it is possible to see string addition – concatenation – in action:

```
10 INPUT "Last name:";last$
20 INPUT "First name:";first$
25 hello$="Hello "+first$+""+last$
30 PRINT hello$
```

Notice the way the semicolons in the

Basically...

WHEN rounding takes place, as in maths, if the fractional part is equal to or greater than 0.5 the number is rounded up, otherwise it is rounded down.

However, there is another system called truncation, here the fractional part is chopped off regardless of its size. Some statements, like INT, perform truncation, while others round, so it is important to recognise the difference.

Real programming

print statement have been replaced by additions for the string in line 25. The output from the program is the same, although the listing is slightly longer. But, if you wanted to print the string *hello$* at different points in the program this way is shorter and simpler. Strings are always added from left to right.

Getting out the scissors

STRINGS of characters, be they names, sentences, words or whatever, when held in variables can be treated very much like real pieces of string. You've already seen how to tie them together and the table below shows the functions for chopping them up, plus a few others.

The most commonly used functions are LEFT$, RIGHT$ and MID$. Type in and run the following to see them in action:

```
10 a$="Left"+"Middle"+"Right"
20 PRINT "Left word=";LEFT$
(a$,4)
30 PRINT "Right word=";RIGHT
$(a$,5)
40 PRINT "Middle word=";MID$
(a$,5,6)
```

Which prints:

```
Left word=Left
Right word=Right
Middle word=Middle
```

Basically...

KEYWORDS which produce a result or value are known in Basic as functions. They are always preceded by the equals sign, for instance =LEFT$ or =LEN. String functions have a dollar appended to their name. All other functions return some numeric value.

Why this happens may not be immediately apparent, so let's examine the listing step by step – something you'll have to do with your own programs when they do odd things.

At line 10 the string variable *a$* is set to *LeftMiddleRight* using concatenation. Line 20 sees the first of the new functions – LEFT$. Its syntax is:

```
a$=LEFT$(b$,X)
```

where the *a$* is the leftmost *X* characters of the string *b$*. The string itself may be either enclosed by quotes, a string variable, or a function returning a string value. Similarly, the numeric value *X* may be a number, a numeric variable, or a function returning a number. Negative values of *X* will cause an error.

Line 30, is exactly the same as LEFT$ except that RIGHT$ returns the right side of the string. Line 40 adds an extra dimension to string handling. Here a third parameter has been added and the syntax of MID$ is:

```
a$=MID$(a$,start,extent)
```

Here *start* is a number pointing to the position in the string to start reading from and *extent* is the number of characters to actually read. The rules appertaining to RIGHT$ and LEFT$ still apply.

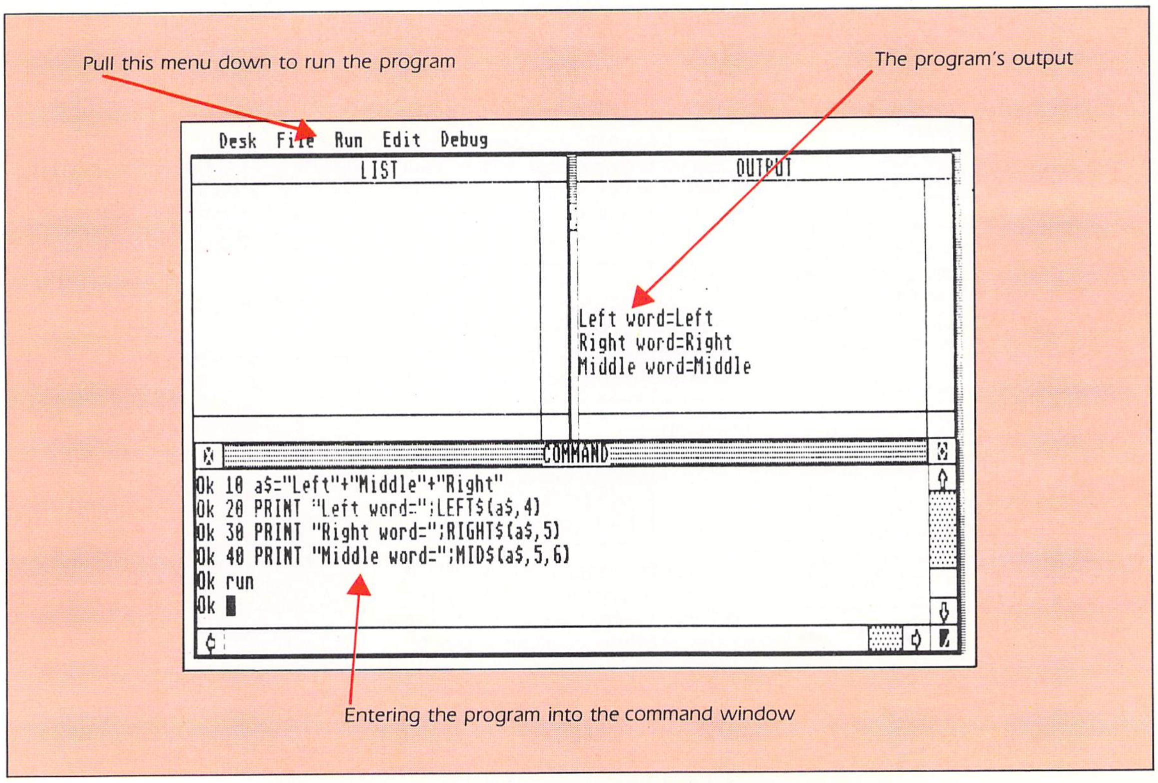
In the example, starting from position five and going on for six characters yields the word Middle.

Function	Operation
LEFT$	Get left part of a string
RIGHT$	Get right part of a string
MID$	Get middle of a string
LEN	Find length of a string
INSTR	Search one string for another
VAL	Return the numeric value of a string
STR$	Convert a numeric value to a string

ST Basic string operations

Try this ...

● Write a program to add three floating point numbers, double the result then print it as a floating point number, a rounded integer and a truncated integer.
● Write a program to approximate PI (22/7), then use it to calculate the area of a circle of (PI*radius^2).
● Write a program to input a two strings, concatenate them and print the result.
● Modify the last program to print out the two halves of the resulting string (use LEN(string$)).


Pull this menu down to run the program
The program's output
Desk File Run Edit Debug
LIST
OUTPUT
COMMAND
Entering the program into the command window

```
Left word=Left
Right word=Right
Middle word=Middle
```

```
Ok 10 a$="Left"+"Middle"+"Right"
Ok 20 PRINT "Left word=";LEFT$(a$,4)
Ok 30 PRINT "Right word=";RIGHT$(a$,5)
Ok 40 PRINT "Middle word=";MID$(a$,5,6)
Ok run
Ok
```

Looping the loop

Repeating yourself

COMPUTERS are well known for calculating problems at breakneck speed. The ST is of course no exception, but up until now we've only seen how to make Basic do drainpipe arithmetic. That is, the program falls straight through from the top to the bottom – one line at a time – like a marble in a drainpipe – until it reaches the end. This sort of thing is probably simpler to do on a pocket calculator, so what is the advantage of Basic?

More often than not, certain operations need to be computed more than once. Consider the case of a simple football pools program. Now let's assume the team names will be entered one at a time. The program has to do the same thing – ask for a team name – many times. Writing each instruction on a separate line would take forever, but Basic has a simple answer – loops.

There are two types of loop in Basic programming simple and controlled. Of the two, simple loops are most often used to keep a program running without end – like an arcade game. Simple loops are best performed with the much maligned GOTO statement. In fact, in ST Basic GOTO is the only way to perform such a function. Try this:

```
10 PRINT "Buy Atari ST User";
20 PRINT ". It's great!"
30 GOTO 10
```

When you run this program, it won't stop. This is because Basic is in an uncontrolled loop – continuously executing the three lines. When Basic reaches line 30 the GOTO statement tells Basic to go straight back to line 10. Stop the program by selecting Break from the Run menu. In fact, you can tell Basic to go to any line number in a program, however, forward jumps are not recommended.

Basically...

ALL FOR ... NEXT loops have three parts. A head – where the loop is defined. The body – the part that's repeated – and the tail where the loop is tested and repeated if necessary. Other looping constructs like WHILE WEND are tested at the head, the tail simply closes the loop.

Avoiding knots

BY its nature GOTO should only be used where absolutely necessary otherwise you're liable to tie yourself into an inextricable software knot. Consider the following:

```
10 GOTO 40
20 GOTO 50
30 PRINT "Here!"
40 GOTO 20
50 GOTO 30
```

What happens when you run this? As you can see GOTO has now tied this program around itself. While this may seem silly imagine a 20k long program with similar jumps. Frightening isn't it? The program may work – but what if? Finally, never jump out of a controlled loop. You may be tempted, but you'll just create needless confusion.

Taking control

THE most commonly used controlled loop in Basic is the FOR ... NEXT construct – it's also the simplest form of closed loop. Now for an example, type in and run the following:

```
10 FOR n=1 TO 10
20 PRINT "Number=";n
30 NEXT
40 PRINT "Final=";n
```

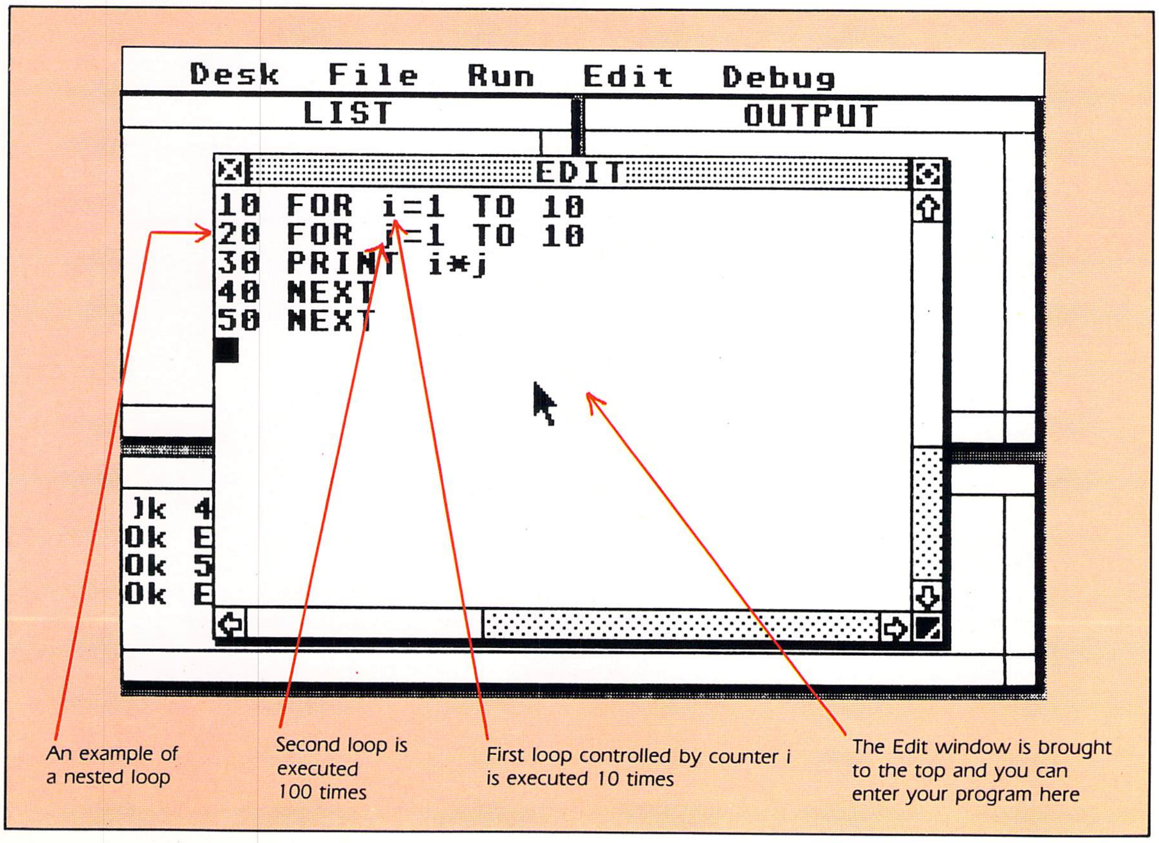
As you will see, Basic counts from 1 to 10 and prints the final result. This simple program is split into three distinct parts. We'll now examine each one in turn. (The panel shows a more detailed explanation of FOR ... NEXT).

● Line 10 is the start of the loop, and here we tell Basic to start counting at one and stop at 10. The loop variable or counter is *n*. The optional STEP statement has been omitted, since in this case the default increment of one was required.

● Line 20 forms the body of the loop. This prints the current contents of the loop variable, *n*.

● Line 30 forms the tail. This is where the loop variable is incremented by one and tested against the limit. If this is exceeded the loop terminates.

● Line 40 prints the final value. Note this is one higher than the number of


Desk File Run Edit Debug
LIST
OUTPUT
EDIT
An example of a nested loop
Second loop is executed 100 times
First loop controlled by counter i is executed 10 times
The Edit window is brought to the top and you can enter your program here

```
10 FOR i=1 TO 10
20 FOR j=1 TO 10
30 PRINT i*j
40 NEXT
50 NEXT
```

Looping the loop

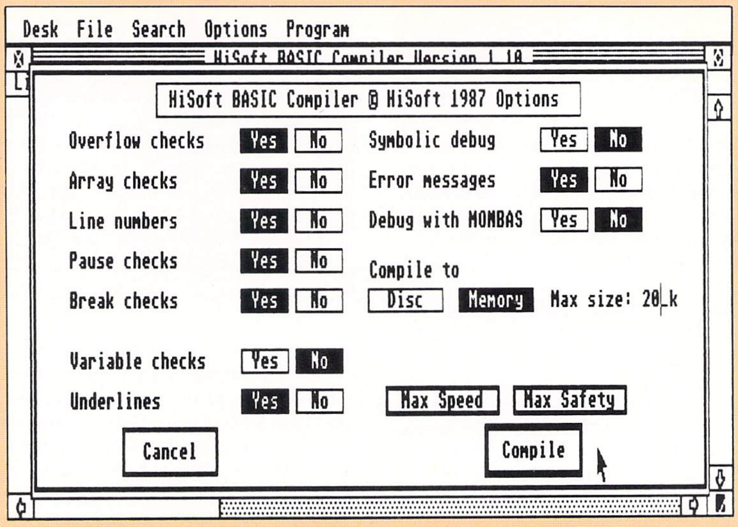
Basically...

IN any program, a loop is any block of instructions which is executed more than once. There are two types of these – the open loop and the closed loop. Open ones execute a variable number of times and closed loops execute a fixed number of times.

loop counts (iterations). But what if you want to count backwards? Replace line 10 with this:

```
10 FOR n=10 TO 1
```

Now run the program. It doesn't work, but why?



Desk File Search Options Program

HiSoft BASIC Compiler Version 1.10

HiSoft BASIC Compiler © HiSoft 1987 Options

Overflow checks [✓ Yes] No
Array checks [✓ Yes] No
Line numbers [✓ Yes] No
Pause checks [✓ Yes] No
Break checks [✓ Yes] No
Variable checks Yes [✓ No]
Underlines [✓ Yes] No
Symbolic debug Yes [✓ No]
Error messages [✓ Yes] No
Debug with MONBAS Yes [✓ No]
Compile to Disc [✓ Memory] Max size: 20 k

HiSoft Basic compiler will load and compile ST Basic listings, enabling you to create stand-alone programs that execute directly from the Gem desktop.

Stepping out

IN the previous example we assumed the step size was one so the program worked, however this time Basic is trying to count backwards from 10 to 1 with a step size of +1. In other words the step size was wrong so it skips the loop body. To correct this problem it is necessary to use a negative step size, which is achieved like this:

```
10 FOR n=10 TO 1 STEP-1
```

Similarly to count from positive numbers to negative numbers using a larger increment use the following:

```
10 FOR n=10 TO -10 STEP -2
```

Notice here there are only 10 loops.

One of the best features of controlled loops is they can be nested. Nesting is a process where one controlled loop becomes the body of a second or even third. There is a limit to how deep such loops can be nested but I've never heard of anyone reaching it by design. Here's an example of a nested FOR ... NEXT loop:

```
10 FOR i=1 TO 10
20 FOR j=1 TO 10
30 PRINT i*j
40 NEXT
50 NEXT
```

The body of the first loop controlled by the counter *i* is executed 10 times, similarly the body of the second inner loop is executed 100 times. I'll leave you to determine why this is for yourself.

Try this. . .

● Write a program to print out the nine times table, with the results formatted:

```
1 * 9 = 9
2 * 9 = 18
```

● Use a GOTO to determine the maximum level of nesting of FOR ... NEXT loops. This will create an error to watch out for in your own programs.

● The NEXT statement can be optionally followed by the name of its control variable. What happens if the NEXTs of two nested loops are mixed up?

FOR ... NEXT loops

THE syntax of the FOR ... NEXT loop is as follows:

```
FOR count=start TO limit
STEP stepsize
(statements)
NEXT count
```

Here the head of the loop contains the keyword FOR, and this marks the start of the loop. The next stage contains the variable assignment:

```
count=start
```

This variable *count* – known as the loop variable – must be supplied and can be any valid numeric variable. However, in many cases an integer variable would be used. The numeric value *limit* can itself either be a number or a variable. This will be the number the loop will start counting from.

Next comes the keyword TO. This is an essential part of the syntax, although it does little other than aid readability.

Finally, the last essential part of the FOR syntax is *limit*. Once again this can be any valid numeric variable or just a number. This is the limit of the loop.

An optional part of the FOR ... NEXT construct is STEP. This determines the size of increment – that is the amount added to the loop variable when the NEXT statement is reached.

The variable *stepsize* can be either a variable or a number. Basic assumes a step size of one so this statement is normally omitted for simple loops.

The number of times the loop is executed therefore is given by:

```
(limit-start)/stepsize
```

The loop body follows the FOR header. This is the set of Basic commands, functions or statements – a mini program if you like – that will be repeatedly executed. The loop body is also optional, leading to the fact that FOR ... NEXT is often used for time delays.

Finally, comes the NEXT statement. This can optionally be followed by the loop variable for clarity, although normally this is left off. When NEXT is reached, Basic adds *stepsize* to *count* and checks to see if *limit* has been exceeded. If it has, the loop terminates immediately, if not, control returns to the first statement after the head.

Conditional programming

Decisions and more decisions

AS human beings, we make decisions all the time – some trivial, some important. For instance, is lager cheaper at the Black Horse or the White Rabbit? According to a set of rules, or determining factors, we make a rational choice. The Black Horse is cheaper but the White Rabbit is closer – we'll go to the Black Horse.

Computers of course, can't think in such terms, so we are reduced yet again to a case of pure logic, courtesy of binary electronics. Basic of course makes things a little simpler, and instead of thinking in ones and zeros, Basic thinks in terms of true and false. From a simple numerical assumption, it can draw a conclusion whether it is true or not – and act upon that decision.

This ability to decide on one course of action or another, forms the backbone of all programming in every programming language. To see how Basic reacts to pure logic type in and run the following listing:

```
10 A=1:B=2
20 PRINT "A equals B:";A=B
30 PRINT "A not equal to B:";A<>B
40 PRINT "A is biggest:";A>B
50 PRINT "B is biggest:";A<B
```

Basically...

WHEN ST Basic meets an IF statement it will expect to find one or more conditions and a THEN statement telling it what to do next. If the test fails the optional ELSE part will be executed. Afterwards program control falls to the next line in sequence unless redirected by a GOTO.

Basic responds with:

```
A equals B:0
A not equal to B:-1
A is biggest:0
B is biggest:-1
```

This example raises some important points, not least what the symbols mean. They're known as the relational operators, and a full list appears in the table below.

The designers of Basic borrowed what symbols they could from algebra, so some may be familiar. Certain symbols however, like algebra's crossed equals, are not available on most micros (or mainframes come to that) so the designers invented their own.

=	... Equal to
<>	... Not equal to
>	... Greater than
<	... Less than
>=	... Greater than or equal to
<=	... Less than or equal to

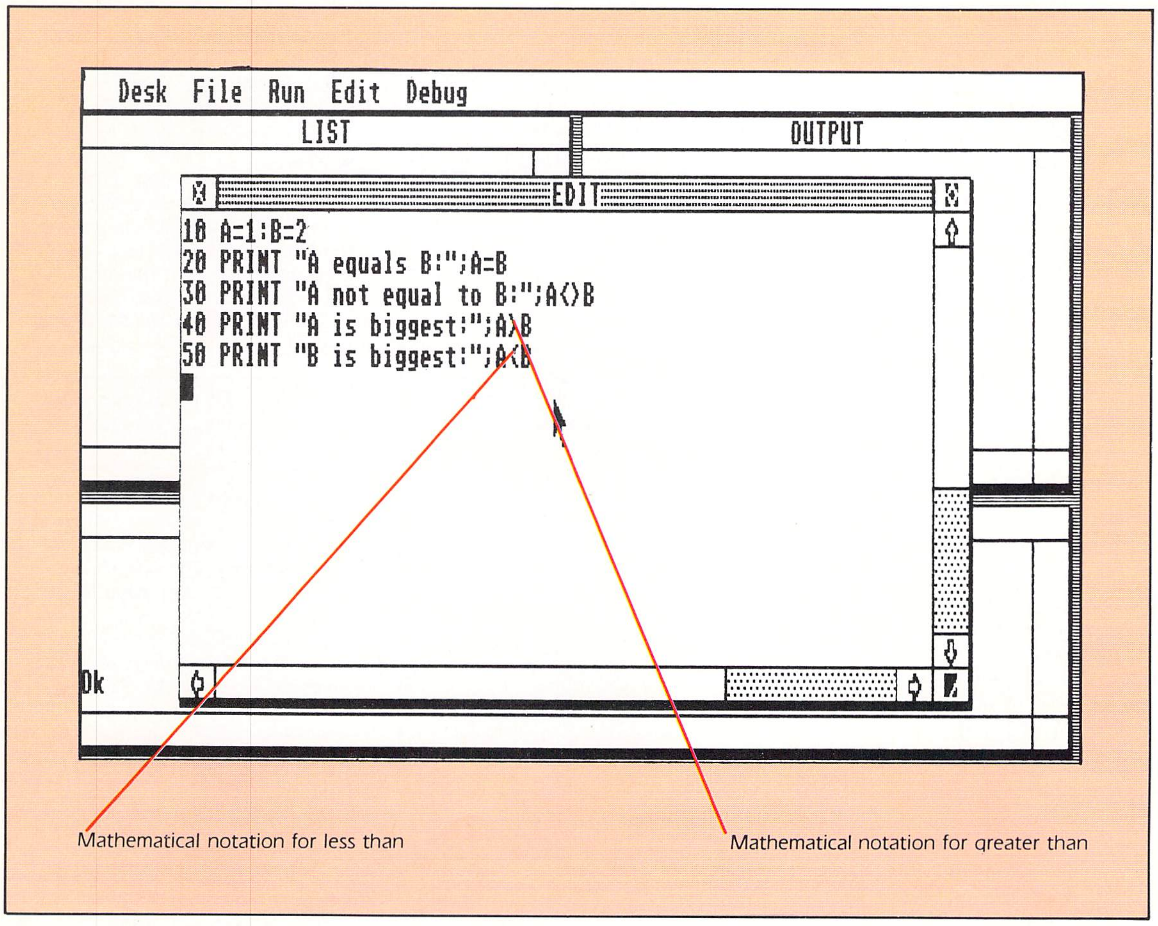
ST Basic's relational operators

Two of the above statements are true and two are false – indicated by a result of true (–1) or false (0). By studying the program it should be clear which are true and which aren't. The two values are not the same and B is the largest.

Basic's lie detector

WHAT happened was that Basic was drawing conclusions from a premise. Given A=1 and B=2 and then told to print A=B, it concluded that A=B was false and printed zero to indicate this. Similarly, when asked to print A<B, as A is less than B, Basic concluded this statement was true and printed –1. Read the listing again and it will soon become clear what is happening.

This is how Basic always arrives at its conclusions. No matter how complex a logical expression may appear, Basic breaks it down (parses it) into manageable chunks and checks the validity of each one in turn. The value –1 is used internally by ST Basic to represent a true conclusion and 0 to represent


Desk File Run Edit Debug
LIST
OUTPUT
EDIT
Ok
Mathematical notation for less than
Mathematical notation for greater than

```
10 A=1:B=2
20 PRINT "A equals B:";A=B
30 PRINT "A not equal to B:";A<>B
40 PRINT "A is biggest:";A>B
50 PRINT "B is biggest:";A<B
```

Conditional programming

false. However any none-zero value is treated as true and zero is always false. I'll explain how this can be of use later on.

Testing, testing

AT last we arrive at the crossroads in Basic programming – the decision makers themselves. It is essential that in a program you are able to test a variable or expression and see whether it is true or false.

For instance, suppose you are writing a game. One of the most mundane tasks is writing the so-called user interface. This is the part which allows you to move say, your spaceship and fire missiles to blow up the enemy.

Without some sort of testing facility, it wouldn't be possible to discover which way the joystick was facing or whether the fire button was pressed. Or, on a more down to earth level, how do you check the values from an INPUT to make sure they stay within predefined limits the program can handle?

Basic provides the tool for all this and more in the misleadingly simple form of the IF THEN ELSE statements. The syntax of these is either:

```
IF <condition> THEN <statement>
```

Or with the optional ELSE:

```
IF <condition> THEN <statement1>
                ELSE <statement2>
```

Before delving into the workings of these take a look at the following simple example:

```
10 FOR N=1 TO 10
20 PRINT N
30 IF N=5 THEN PRINT"Half way
through"
40 NEXT
```

This program is quite straightforward in operation but it shows IF ... THEN in operation at its simplest level as a straightforward test. The message "Half way through" is only printed when the value of *N* is exactly five. Now alter line 10 to read:

```
10 FOR N=1 TO 10 STEP 3
```

When the program is run with this alteration the message is not printed. This is because the value of *N* never actually equals five.

By now you should be able to see how IF works in its simplest form. When the line is executed, Basic does a test on the condition to see if it returns true or false (0 or –1). If the condition is satisfied – true – everything following THEN is executed. If not, execution continues on the next highest line number in sequence.

A more useful use of IF is in range checking inputs. Remember the problem above?

Try the following:

```
10 INPUT"Enter any number up
   to 10";n
20 IF n>10 THEN PRINT "Too b
   ig, try again":GOTO 10
30 PRINT "You entered";n
```

When this program is run, and a number greater than 10 is input a message is printed and you are asked to try again. Note how the GOTO statement follows immediately after the PRINT and is separated by a colon. This is because we only want to return to the input – at line 10 – if an error occurred.

Any statements placed after the GOTO would never be executed and would therefore be meaningless. There is of course one exception to this rule – the ELSE statement, more of this later.

The previous example works perfectly well until your program requires a range of numbers, say between 0 and 10. Remember that numeric variables can hold negative numbers as well as positive ones. If you enter a negative number where a positive value is required the program will in all probability produce an erroneous result.

In this case you could use the ABS function, which makes all numeric values positive, but this may also cause an error. Alternatively you could use a second IF ... THEN to trap any value less than zero by adding the following line:

```
25 IF n<0 THEN PRINT"Too sma
ll, try again":GOTO 10
```

Now this is all well and good, but it isn't very neat. Besides which it may not be necessary to give a message explaining the fault in great detail. A message such as "Out Of Range" will probably suffice, if indeed you want to give any message at all.

The simple answer then, is to combine the results of the two conditions into one logical statement. Once again, before exploring how these work remove line 25 and amend line 20 to:

```
20 IF n<=-1 OR n>=11 THEN PR
INT"Out of range, try again"
:GOTO 10
```

By way of a little variety, I've used some alternative relational operators to perform the actual range testing. Referring back to the operator table should clarify these and serves to demonstrate how they can often be interchanged to suit your own personal preference. Or, when you become more experienced, actually make a program more readable to others.

Try reading the last example out loud, replacing the symbols with their meaning. This should demonstrate the thinking behind the test and what happens. As a good rule of thumb: If you can't read it out loud it's too complex. In this case the line is simplest thus:

```
20 IF n<0 OR n>10 THEN PRINT
"Out of range, try again"
:GOTO 10
```

By now, you may well be wondering what the OR does, though I bet many of you have guessed. Consider the table alongside. This is what is called a truth table for the conditional OR operator.

If you've never seen one before, it can look a little daunting although it is quite easy to understand once you get the hang of it.

Once again, it has its roots very firmly planted in binary electronics and a mathematical technique known as boolean algebra.

Fortunately, you don't need to know anything about these subjects to understand their usage in Basic. Logical operators always take two values organised like a normal arithmetic statement.

True OR false?

WHEN Basic performs a logical test it leaves a result of true or false as a conclusion. This is where the truth tables come in. Using the truth tables as a guide, we can combine the results of two or more true/false tests to come out with an overall answer of true or false. To make this clearer let's examine the revised line 20 on the right with *n* set to 3:

- IF .... Start of the IF .. THEN test.
- n<0 .... n is less than 0? False.
- OR .... The logical operator.
- n>10 ... n is greater than 10? False.
- THEN ... Considering the truth table for OR we find False OR False is False so the test fails so the rest of the line is ignored.

Now let's consider what happens

A OR B	=	Result
False OR False	=	False
True OR False	=	True
False OR True	=	True
True OR True	=	True

The OR truth table

if you enter a value of 12 for *n*:

- IF .... Start of the IF THEN test.
- n<0 .... n is less than 0? False.
- OR .... The logical operator.
- n>10 ... n is greater than 10? True.
- THEN ... This time False OR True is True. The test succeeds and the statements following THEN are executed causing the error to be printed and sending control back to the input at line 10.

The truth of the matter

Logical operators

SO far we have looked at the OR logical operator, but as you can see from the tables alongside, this is not the only one. In fact all the operators listed here can be used in much the same sort of way as we'll see, but OR and AND are probably the most common and useful.

Comparing the truth tables for AND and OR indicates they have very little in common, although in fact OR includes the true result of the AND function. True OR true gives a result of true, also true AND true is true.

Let us assume for a moment, that two values – which we'll call *high* and *low* – need to be checked, and both must be out of range before reporting an error. Writing this in terms of OR won't work. Look at the following:

```
IF low<0 OR high>10 THEN PRINT
"Out of range, please try again"
```

If this line was part of a program and the two values were in range it would work – that is, the message would not be printed. If however, just one value strayed outside the specified limits, an error would be reported, and this isn't what is wanted. More confusingly, because the OR operator includes AND, the error would be flagged if both values were also incorrect.

The line would be better written in terms of AND:

```
IF low<0 AND high>10 THEN PRINT
"Out of range, please try again"
```

Now only if both conditions are satisfied will the message be printed. The difference is subtle admittedly, but there all the same.

There are times, of course, when an OR is required, but the AND function which it includes would cause problems. In other words, we want to do something when only one of two conditions is true. The OR operator would seem the logical choice, but it will not always work.

AND	...Logical AND
OR	...Logical OR
XOR	...Logical Exclusive OR
NOT	...Logical NOT
IMP	...Logical implication
EQV	...Logical equivalence

ST Basic's logical operators

False AND False= False
True AND False= False
False AND True= False
True AND True= True

The AND truth table

NOT true= False
NOT False= True

The NOT truth table

False XOR False= False
True XOR False= True
False XOR True= True
True XOR True= False

The Exclusive OR truth table

False IMP False= True
True IMP False= False
False IMP True= True
True IMP True= True

The implication truth table

False EQV False= True
True EQV False= False
False EQV True= False
True EQV True= True

The equivalence truth table

Truth tables for the logical operators

Suppose we want the error reported if one or the other condition is true, but not both. In this case OR simply won't do, so instead the XOR operator is used. XOR (eXclusive-OR) gets its name because it excludes the AND function from OR:

```
IF low<0 XOR high>10 THEN PRINT
"Out of range, please try again"
```

Surprisingly, XOR is very rarely needed in practise, as OR will usually suffice.

There will be times when more than one logical statement will be required. As logical operators have a priority in the order NOT, AND, OR, XOR, IMP, EQV, this can sometimes cause ambiguous results. Although it won't usually cause problems, try and work out the truth or falsehood of the following:

```
10 a=1:b=2:c=0
20 PRINT a=1 OR b=2 AND c=3
```

Depending on which way this is approached will determine whether or not –1 (true) or 0 (false) is printed. A better way of making sure your meaning is clear is to enclose some of the operations in parentheses. In other words, break up the line into easily understandable chunks and make the meaning clear. This will also result in making the program easier to read later on. For instance:

```
10a=1:b=2:c=3:d=0
20 PRINT (a=1 OR b=2) AND c=3
```

has a different result to:

```
10a=1:b=2:c=0
20 PRINT a=1 OR (b=2 AND c=3)
```

When the test fails

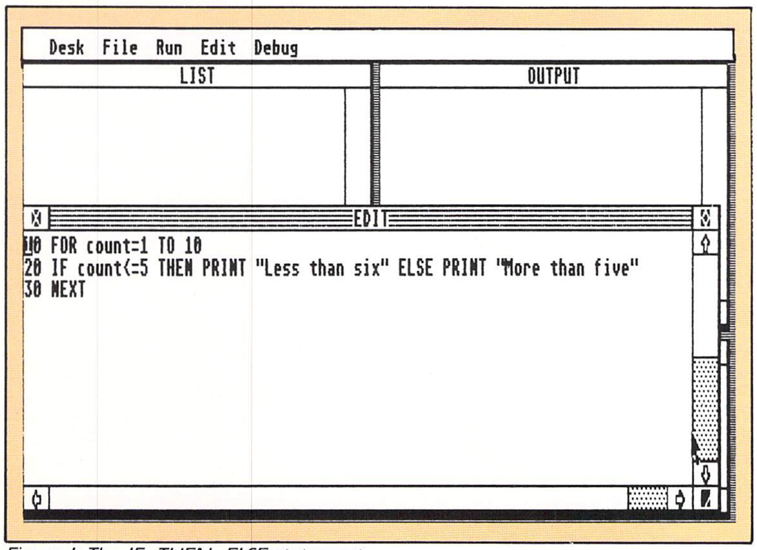
THERE are, of course, instances when a certain set of conditions calls for one action or another. The most obvious method would be to use a second or even third conditional test. However, ST Basic provides a rather more elegant solution in the IF ... THEN ... ELSE statement.

The commands following the ELSE are executed whenever the conditional part of the IF fails. Since ELSE is optional to the IF THEN syntax, if it can't be found, program control resumes at the next line number in sequence. To see this in action enter and run the following:

```
10 FOR count=1 TO 10
20 IF count<=5 THEN PRINT "Les
s than six" ELSE PRINT "More t
han five"
30 NEXT
```

As soon as *count* reaches six the test fails and the ELSE part of line 20 is executed. It's as simple as that. In fact, the ELSE can be followed by a further IF THEN if necessary. However, use of a second ELSE may make the line ambiguous to you and other versions of Basic. Avoid it.

One last point worth mentioning is



```
10 FOR count=1 TO 10
20 IF count<=5 THEN PRINT "Less than six" ELSE PRINT "More than five"
30 NEXT
```

Figure I: The IF...THEN...ELSE statement

The truth of the matter

Basically...

WHEN using relational operators to make tests it is important to note that we make a statement to Basic, and it decides the validity of that statement. This may sound a roundabout way of doing things, but if you think in this way, it's often easier to construct and debug lengthy conditional statements.

the fact that Basic uses zero and non-zero values to indicate true or false. This can be used to produce some simple statements. It is most often used for a system known as flagging – setting a numeric variable to indicate truth or falsehood.

A flag can be any spare numeric variable, although it helps to give it a meaningful name. If you were writing an arcade game for instance, the end of the game could be influenced by many different factors – normally the player losing all his lives. In the main loop, a flag can be tested to indicate this – even though the flag could be set at several different points in the program. Here's how to do it:

```
IF dead THEN PRINT "Game over"
:END
```

Initially, when the program is run *dead* is set to zero, so the test always fails. When the player loses all his lives though, for whatever reason, *dead* can be set to any non-zero value.

Implying something else

AFTER this heady discussion, there still remain two logical operators to be explored – EQV and IMP. By their nature they're not often used in programming, but find a use in specialised applications. Both allow a program to draw conclusions about something given a relationship between two items.

The EQV (equivalence) operator checks two logical statements or conditions and leaves a logical true if they are the same. This relationship can be seen in the truth table. For instance try the following:

```
10 PRINT 2+2=4 EQV 3+3=6
20 PRINT 3+2=2 EQV 5+5=1
```

Similarly, the implication operator IMP checks two logical statements to see if the conclusion (left of the IMP) is justified by the premise (right of the IMP). If so IMP leaves a logical true as the result. Now try this, remembering the first statement is the logical premise:

```
10 PRINT 2+2=4 IMP 3+3=6
20 PRINT 2+2=4 IMP 3+3=7
30 PRINT 2+2=5 IMP 3+3=6
40 PRINT 2+2=5 IMP 3+3=7
```

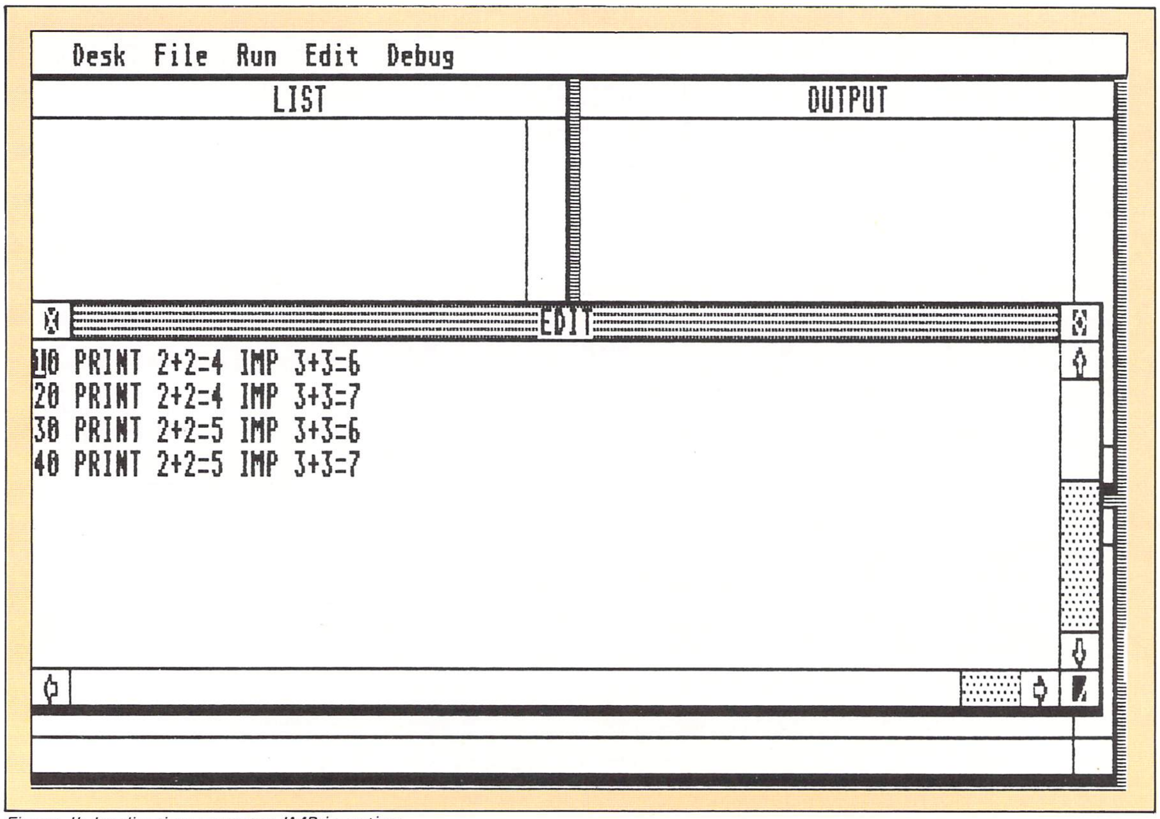
As an exercise I'll leave you to discover why these programs arrive at their conclusions. But don't be put off by the apparent complexity of all of this. Most programmers only ever use AND, OR and NOT; the others are merely there for completeness.

Basically...

THE use of ELSE after an IF ... THEN to call an unconditional branch via GOTO should be avoided – especially if the THEN statement already involves a branch. Not only is this bad programming practice, it can also lead to great confusion.

Basically...

PROBABLY the simplest and least used of all logical operators is NOT. This simply reverses the truth of a logical statement, and proves that black can be white. It can often be used to simplify an IF ...THEN...ELSE statement by reversing the order of execution and removing the need for ELSE.



```
10 PRINT 2+2=4 IMP 3+3=6
20 PRINT 2+2=4 IMP 3+3=7
30 PRINT 2+2=5 IMP 3+3=6
40 PRINT 2+2=5 IMP 3+3=7
```

Figure II: Implication operator IMP in action

While on the other hand . . .

Simulating REPEAT?

THE FOR ... NEXT loop construct is the simplest form of controlled loop, but there are several other types too. Some Basics, like STOS, include the REPEAT ... UNTIL construct, which repeats several program lines until a certain condition is found to be true.

Others – and this includes ST Basic – use the similar but not quite identical WHILE ... WEND construct to repeat part of a program while a certain condition is true. However, REPEAT ... UNTIL can be simulated in ST Basic quite easily should you require it by using IF *x* THEN GOTO *repeat*.

Like FOR ... NEXT, a WHILE ... WEND loop has three distinct and separate parts – a head, body and tail. The WHILE statement forms the head of the loop, the optional body forms a series of commands or lines to be repeated, and the WEND forms the tail.

This is where the similarity ends. Although this is a controlled loop structure, the loop does not terminate until a certain condition – defined in the WHILE part – fails to be met. The syntax of WHILE ... WEND is as follows:

```
WHILE <condition is true>
  execute program lines
WEND
```

As you can see, this new loop structure appears to be very simple, although it does introduce a feature borrowed from IF ... THEN – the conditional test. Unlike FOR ... NEXT – a loop which executes a definite number of times according to instructions given in the head – WHILE executes either indefinitely until a certain condition ceases to be true, or never at all.

If the condition fails to be met when the head is executed for the first time, program control jumps directly to just after the WEND marking the end of the loop structure. It is therefore, important to realise that WHILE ... WEND can completely ignore the statements in the loop body if certain conditions occur.

WHILE or REPEAT?

LOOK at this example to see how WHILE loops differ from REPEAT loops. (Remember, ST Basic does not include REPEAT and it is being featured here to illustrate the very similar WHILE loop).

Here is WHILE:

```
10 count=0
20 WHILE count<>10
30   count=count+1
40 WEND
50 PRINT count
```

And the REPEAT construct – if ST Basic had one – would look like:

```
10 count=0
20 REPEAT
30   count=count+1
40 UNTIL count=10
50 PRINT count
```

These two examples perform exactly the same task. Even so, in this case it would be easier to use a FOR ... NEXT construct as all the program does is perform a simple count incrementing from 0 to 10.

However, you should notice an important variation: In the case of the WHILE construct, the condition is tested at the head of the loop, but the REPEAT tests it at the end, after it has executed the loop. If the test fails on the first run of the WHILE loop, the counter would never be incremented, yet the REPEAT loop would.

The following example illustrates this more clearly:

```
10 count=10
20 WHILE count<>10
30    count=count+1
40    PRINT "Counting..."
50 WEND
60 PRINT count
```

Loop structures

ST BASIC has only two loop structures, the fixed FOR . . . NEXT and the more flexible WHILE . . . WEND statements. HiSoft and Power Basics – both by HiSoft – are completely compatible with ST Basic, so you can type in and run all the listings here. However there are several advantages in using these alternative Basics over ST Basic.

The first advantage is that the programs are compiled directly to machine code and therefore run hundreds of times faster. Secondly, a number of ST Basic bugs are fixed, and finally there are many additions and extensions to the range of commands available. For instance, as well as the two ST Basic loop structures two others are available in HiSoft Basic. The first is the REPEAT . . . END REPEAT loop:

```
i = 0
REPEAT one
  i = i + 1
  IF i = 10 THEN EXIT one
  PRINT i
END REPEAT one
```

The REPEAT statement is followed by a variable, in this case one. *The end of the loop is marked by END REPEAT. Unless you break out of the loop with an EXIT command – here shown with an IF . . . THEN statement – then the loop will repeat forever.*

The second loop structure is the very flexible DO . . . LOOP which can be used in a variety of ways. You can DO . . . LOOP UNTIL, DO UNTIL . . . LOOP or simply DO . . . LOOP. The following three snippets of HiSoft Basic code show how they are implemented:

```
i=0
DO
  i=i+1
  PRINT i
LOOP UNTIL i=10

DO UNTIL i=10
  i=i+1
  PRINT i
LOOP

i=0
DO
  i=i+1
  IF i=10 THEN EXIT LOOP
  PRINT i
LOOP
```

The UNTIL statement tests to see if a certain condition is true – in this case if the variable i *is equal to 10 – and exits the loop when it is. The last DO . . . LOOP hasn't got an UNTIL so it will loop forever unless an EXIT LOOP is met.*

It's worth noting that if you do decide to use this enhanced version of Basic you can't subsequently run the programs in ST Basic, as the extra commands won't be recognised. However HiSoft and Power Basics do recognise all ST Basic (old version) commands.

While on the other hand . . .

Here the WHILE skips the body of the loop and jumps straight to the end where it prints the value of *count*. Notice that the message in the body of the loop is never printed.

Data handling

THERE are times while writing a program that you need to store a series of constant data items. Take, for example, a program which prints the name of any month given just the month number (1 to 12) to go on. How do you get the names to print?

One way would be to implement a series of IF ... THEN tests on the month number and then print the correct month's name. Or you could store the names in string variables. You are then faced with picking which string to print like this:

```
10 Month1$="January"
20 Month2$="February"
30 Month3$="March"
.
.
.
200 IF n=1 THEN PRINT Month1$
210 IF n=2 THEN PRINT Month2$
220 IF n=3 THEN PRINT Month3$
```

This may seem like the obvious way of solving the problem, but of course, there are alternative methods. Basic has a far neater way around this problem, which crops up an awful lot in the real world.

Basically...

THE only major difference between the WHILE loop and the REPEAT loop is that the latter always executes at least once. Otherwise the choice between the two functions is arbitrary. ST Basic does not support REPEAT, but more advanced Basics like STOS do.

Either function can be simulated with IF *x* THEN GOTO *label*, but this should be avoided, as GOTOs tend to lead to programs that are complex and difficult to debug.

Here is another example: The parts department in a garage stores spare parts for a particular type of car. The number of parts is constant, and so is each part number. The store manager wants to computerise the system so that he can enter a part number and the micro will print out the part name (or vice versa).

The problem is essentially the same as before – inputting a number and then printing a string as a result. This time however, there are a much greater number of items so an enormous number of IF ... THEN statements would be required.

Basic has two features to solve this problem, and breaking the task down further you may realise there are two things featured here. First there is a long list of constant values, and second there is a need to store all of these values inside the program.

Solving this problem using variable assignment and IF ... THENs as in the calendar example would take up massive amounts of memory for a stock list and would, therefore, be completely unsatisfactory. There is a much better way, and that is by tying all of the similar variables together in the form of a list, or in Basic terminology, an array.

A new dimension

ARRAYS are just special forms of the variables we have already discussed. In fact, any variable can be defined as an array using the Basic statement DIM – short for dimension.

What this means is that one variable name can have many different values. Each value is called an element and is accessed by an index number which is given in brackets when the variable is accessed, that is, assigned or read.

When the program is run, Basic reserves memory for the elements belonging to the array defined in the DIM statement. This is only performed once in the program and attempting to do it again gives rise to an error.

To see DIM in action try:

```
10 DIM numbers(10),string$(10)
20 FOR i=1 to 10
30   PRINT numbers(i),
     string$(i)
40 NEXT
```

This short program simply allocates space for two arrays. The first is a single precision real and the second a string type. The thing to note here is that DIM initialises all of the elements in a numeric array to zero and all elements in a string array to null or empty strings.

If you want the array elements to hold particular values you must set them in your program like this:

```
10 DIM numbers(10)
20 FOR i=1 to 10
30   numbers(i)=i*i
40 NEXT
50 PRINT "All set!"
60 FOR j=1 to 10
70   PRINT numbers(j)
80 NEXT
```

Once again, what we have done is set up a 10 element array, only this time it has been initialised to hold a set of calculated values – the square of the loop counter. This set of numbers is then displayed on screen using a simple loop.

Our store manager could define a string array for the parts names and a numeric array for the parts numbers. After entering a number, say *n*, the program could print out the part name with a simple PRINT part$(n).

Basically...

ALL types of loop structure can be simulated with IF . . . THEN . . . GOTO, and in fact early implementations of Basic didn't have any looping structures apart from FOR . . . NEXT. This led to so-called spaghetti programming with GOTOs sending program flow careering all over the listing.

The loop structures are designed to prevent this and to make programs more readable and easier to follow. So make sure you use them where possible.

Arrays like this are called single dimensional arrays, but it is possible to have two or even more dimensions. Their number and size is decided by the program, but it is rare to have more than two or three. A nine by nine (two dimensional) array contains 81 (9 x 9) elements, and a two by three by four (three dimensional) array has 24 (2 x 3 x 4) elements.

The most powerful feature of arrays is their ability to hold lists of related data in an easy to get at form. Not only this, they are faster to use than repeating the same calculation, perhaps many times.

This is a technique called tabling and is used widely by computer programmers in all sorts of languages, not just Basic.

Consider the SIN function, which calculates the sine of an angle given its size in radians.

Suppose for example you need to use the sine values, given the number of degrees for a particular calculation. If you require more than just a few calculations, which is quite likely, the fastest method is to set up a table of the sine values from 1 to 90 degrees. Here's how you would do this:

```
10 DIM nums(90)
20 pi=3.141593
30 rad=pi/180
40 FOR i=1 TO 90
50   nums(i)=SIN((i*rad))
60   PRINT "SIN ";i;" = "
     ;nums(i)
70 NEXT
```

A simple solution to a complex problem.

Read all about it

Conversion factors

THERE are times when a program needs to access a lot of pre-defined information. Consider a utility which converts values expressed in one notation to values expressed in another, for instance, converting metric weights and measures into imperial ones. The conversion in itself is quite straightforward – you simply multiply the figure you wish to alter by a conversion factor.

For instance, one inch is approximately 2.5 centimetres, so the conversion factor you would use to change inches to centimetres would be 2.5. A program to convert inches to centimetres would look something like this:

```
10 INPUT "Inches";i
20 PRINT i;"inches = ";i*2.5";cm"
30 END
```

However, the conversion factor for each pair of units like metres-miles, pounds-kilograms and so on is different and there are an awful lot of them, so this program is not a very satisfactory solution. A much better way would be to store all the conversion factors in an array.

But the problem remains: How do you get all the numbers into the array in the first place? The obvious solution is to simply set up the array as a series of assignments:

```
10 DIM fac(5)
20 fac(1)=2.5
30 fac(2)=0.3
40 fac(3)=1.2
50 fac(4)=0.2
```

This works OK, but it causes two problems: First, the program is not very clear, which could give rise to errors, and second it is very long winded to write and debug.

Reading data

WHAT is required then, is some simple way of tying all the data together in a simple and organised fashion – a way you, and anyone else, can understand. Basic fortunately, provides a rather elegant solution to this problem, in the form of the related READ, DATA and RESTORE statements.

The syntax of READ and DATA look like this:

```
READ variable [,variable,...]
DATA item1, item2 ...
```

The READ statement is followed by a list of variables and DATA is followed by a list of data items. At first sight this may look very strange, so let's apply it to a program:

```
10 READ count
20 FOR n=1 to count
30 READ name$(n),title$(n)
40 NEXT
.
.
.
1200 DATA 2
1210 DATA "Mark","Mr"
1220 DATA "Jane","Miss"
```

At line 10 READ attempts to assign the first item of data to *count*. The first data item is the number two, so this is the same as saying:

```
count=2
```

The variable *count* is then used to determine the number of loops to be performed by the FOR ... NEXT structure. READ assigns the following items of data to the arrays *name$* and *title$*. The loop continues until exhausted. At first sight this may still seem a little strange. For instance, how does Basic know where the data is? Or, how does it know what values to put in which variables?

The solution to the first problem is very simple. When Basic comes across a READ statement it searches the program from the start until it comes to a DATA statement. This position is then stored internally in a variable called the data pointer. Every time an element of data is read the pointer is incremented by one. So the data pointer always points to the next item of data to be read – not necessarily on the next line.

The second problem is left entirely to the programmer. When Basic tries to assign data to a variable it assumes the variable and data are compatible, and if not this gives rise to a very confusing error.

Consider the following:

```
10 READ name$,age
20 DATA "Freddy",29
```

This is directly equivalent to:

```
10 name$="Freddy"
20 age=29
```

Basically...

WHILE READ and DATA must exist in the same program, there is nothing to tie them together. They are not a looping structure like FOR ... NEXT or a conditional structure like IF ... THEN ... ELSE. In other words, the DATA keyword can exist anywhere in the program as long as it is the first statement on a line.

Basically...

THERE are several schools of thought regarding where data should be placed in a program. Some say at the start, others at the end, others say as close as possible to the lines where it is used. I prefer to organise data at the end of the program – out of harm's way – where it can be added to or amended easily.

```
  10 REM Metric converter
  20 ready=1:count=0:con$=""
  30 WHILE temp$<>"END"
  40 READ temp$,temp
  50 count=count +1
  60 WEND
  70 RESTORE
  80 DIM con$(count),fac(c
ount)
  90 FULLW 2
 100 FOR n=1 TO count
 110 READ con$(n),fac(n)
 120 NEXT
 130 WHILE ready
 140 GOSUB 170
 150 WEND
 160 REM Print list
 170 CLEARW 2
 180 GOTOXY 0,0
 190 RESTORE
 200 FOR n=1 to count
 210 PRINT "(";n;") ";con$
(n)
 220 NEXT
 230 PRINT:INPUT "Selectio
n";choice
 240 IF choice=count THEN
PRINT "Bye":END
 250 IF choice THEN GOSUB
280
 260 RETURN
 270 REM Subroutine to cal
culate conversion
 280 t%=INSTR(0,con$(choic
e)," ")
 290 from$=LEFT$(con$(choi
ce),t%-1)
 300 t%=INSTR(t%+1,con$(ch
oice),">")
 310 result$=RIGHT$(con$(c
hoice),LEN(con$(choice))-t
%-1)
 320 PRINT:PRINT "Convert:
";con$(choice):PRINT
 330 PRINT "Enter number o
f ";from$;
 340 INPUT from
 350 PRINT from;from$;"=";
from * fac(choice);result$
 360 PRINT:INPUT "Press Re
turn to continue...";a$
 370 RETURN
 380 END
 390 DATA "Cms --> Inches"
,0.394
 400 DATA "Metres --> Feet
",3.281
 410 DATA "Kilometres -->
Miles",0.621
 420 DATA "Inches --> Cms"
,2.54
 430 DATA "Feet --> Metres
",0.305
 440 DATA "Miles --> Kilom
etres",1.61
 450 DATA "END",0
```

Read all about it

which is correct. Now consider this:

```
10 READ name$,age
20 DATA 29,"Freddy"
```

This does not work. See if you can work out why for yourself.

Structuring data

THIS brings us to the third point – organisation. A lot can be said about being neat and structured in programming, but there are few cases when this rule applies so strongly as it does in data. The rules are:

- Keep it simple
- Group related elements together whenever possible

If you follow these guidelines you will find programs are more likely to work first time and are easier to debug if they do go wrong.

Data organisation can be seen in the example listing. This is a program to convert metric measurements to imperial and back again. Don't enter it just yet, but consider the way the data has been arranged:

```
400 DATA "Metres --> Feet",3.281
410 DATA "Kilometres --> Miles",0.621
420 DATA "Inches --> Cms",2.54
```

Each DATA line consists of two items. First there is a text string which is used to form part of an options menu. Alongside it is the conversion factor itself. In this way you can see which conversion applies to which factor. Likewise you can add more conversions very simply. Now let's assume the data had been set out like this:

```
400 DATA "Metres --> Feet"
410 DATA "Kilometres --> Miles"
420 DATA "Inches --> Cms"
430 DATA 3.281,0.621,2.54
```

In this case the program would have to be amended to account for the different layout. All the same, the data does not mean an awful lot – it is just strings of text and meaningless numbers.

Before leaving the subject of data there is one other command, without which most programs would be very hard, if not impossible, to write. As has already been said, the data pointer is incremented by one every time an item of data is read. When Basic runs out of data – the pointer runs past the end of the program – it prints the message *Out Of Data*.

If you want to read the same data more than once you must reset the pointer back to the beginning. This is done with the RESTORE instruction. There are two ways of using it – to start reading the data from the beginning simply use RESTORE, but to start reading at a particular line number use something like RESTORE 2000 where 2000 is the line number.

It is important to remember that you can only restore to the start of a line, but Basic's data pointer can be anywhere along a line.

New sets of data must therefore always start at a new line.

Let's now consider the metric conversion utility. There are two ways of looking at this: Experts would call it contrived, beginners complex and know-it-alls elegant. In fact it has elements of all three. As a program it illustrates many of the points already discussed and brings in a few new ones. Most interestingly though, it is expandable.

It makes use of a simple set of data and a few rules to ensure the data can be expanded easily. When run, it presents a menu of seven items, six of which are the conversions and the seventh a neat exit to Basic. New conversions can be added between lines 440 and 450, and these will be included automatically. You should always aim to make your programs expandable like this if possible.

Here is how it works: Lines 10-60 set up the program and count the number of data elements. Lines 70-80 reset the data pointer and dimension the arrays. Lines 90-150 read the data in to the arrays. Lines 160-260 print the menu of options. Lines 270-370 calculate the conversion. Lines 390-450 contain the data.

The conversion routine makes use of some string slicing to make the program's output more interesting. As an exercise see if you can work it out for yourself.

Handling data

TO further illustrate the use of READ and DATA here is a short program that prints anagrams on the screen. It prompts you to solve it and enter the correct word:

The bulk of the listing is fairly simple, but there are a few lines that may require a bit of thought to work out what is going on. The words are stored in data statements at the end of the listing and they are preceded by a single data item indicating the number of words present.

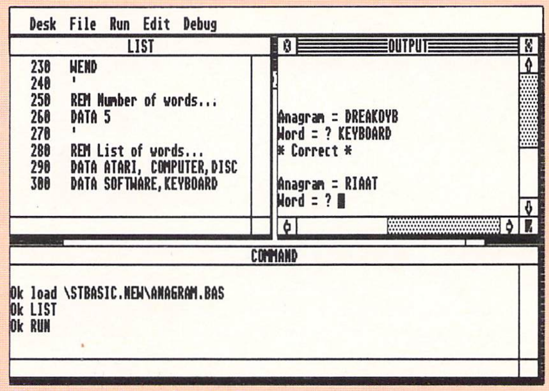
The first task the program performs is to read the number of words into the variable *n%*. Two arrays are then dimensioned – one for the word (*word$*) and one for its anagram (*w$*). A FOR ... NEXT loop reads each word into *word$* and another FOR ... NEXT loop scrambles a copy of it in *w$*.

Lines 100 and 110 are quite complex. The first picks a letter in the word at random and the second puts it at the front of the word. This process is carried out 10 times and the result is that the letters become scrambled.

The WHILE ... WEND loop towards the end of the listing picks a word at random and prints its anagram.

Your guess is compared to the word and the appropriate message is printed via an IF ... THEN ... ELSE statement.

```
 10 REM Anagrams
 20 '
 30 REM Initialise
 40 READ n%
 50 DIM word$(n%),w$(n%)
 60 FOR i%=1 TO n%
 70 READ word$(i%)
 80 w$(i%)=word$(i%)
 90 FOR j%=1 TO 10
 100 p%=1+INT(RND*LEN(w$(i
%)))
 110 w$(i%)=MID$(w$(i%),p%
,1)+LEFT$(w$(i%),p%-1)+MID
$(w$(i%),p%+1)
 120 NEXT
 130 NEXT
 140 '
 150 REM Print anagrams
 160 CLEARW 2
 170 WHILE NOT bored
 180 PRINT
 190 p%=1+INT(RND*n%)
 200 PRINT "Anagram = ";w$
(p%)
 210 INPUT "Word = ";guess
$
 220 IF guess$=word$(p%) T
HEN PRINT "* Correct *" EL
SE PRINT "* Wrong *"
 230 WEND
 240 '
 250 REM Number of words..
 260 DATA 5
 270 '
 280 REM List of words...
 290 DATA ATARI, COMPUTER
 300 DATA SOFTWARE,DISC
 310 DATA KEYBOARD
```



```
230   WEND
240   '
250   REM Number of words...
260   DATA 5
270   '
280   REM List of words...
290   DATA ATARI, COMPUTER,DISC
300   DATA SOFTWARE,KEYBOARD
```

```
Anagram = DREAKOYB
Word = ? KEYBOARD
* Correct *

Anagram = RIAAT
Word = ? █
```

```
Ok load \STBASIC.NEW\ANAGRAM.BAS
Ok LIST
Ok RUN
```

The output from Anagrams

Harnessing Gem

The Graphics Environment Manager on the ST is still a mystery to even the most accomplished programmer. In this section we'll reveal some of its hidden power and demonstrate, with C's flexibility, how to incorporate its friendly user interface into your own programs

The history of Gem

A FEW years ago you had to be a veritable expert if you wanted to use a micro. Even simple operations such as formatting a disc or printing out a file required a considerable and fairly detailed knowledge of computer theory.

At first this was acceptable as the majority of users were enthusiasts who actually enjoyed unravelling complicated procedures. But as time went by people started buying computers for far more practical commercial reasons.

Many users weren't the slightest bit interested in how the computer worked. What they wanted was a system which enabled them to directly apply the computer to their own individual problems with the minimum of effort. Research was therefore conducted to discover new and simpler ways of using computers.

One of the major contributors to this project was Rank Xerox. Its Palo Alto laboratories invented, almost single handedly, the ideas of windows, icons, menus and pointers – wimps – as a way of providing an effective computing environment with a strong visual element. It allowed almost anybody to intuitively operate a computer.

This wimp technology was further developed by Apple into its innovative Lisa and Macintosh computers. Unfortunately it wasn't a great deal of help to the vast majority of business users who were using mainly IBM PCs.

Digital Research eventually came to their rescue with its graphic environment manager, Gem. For the first time this gave PC users a powerful wimp system very similar to that supported by Apple.

Later, when Jack Tramiel was presiding over the development of his new 68000-based computer, he decided to make things easy for potential customers by incorporating a version of Gem into the machine, and the Atari ST as we know it was born.

Gem on the ST

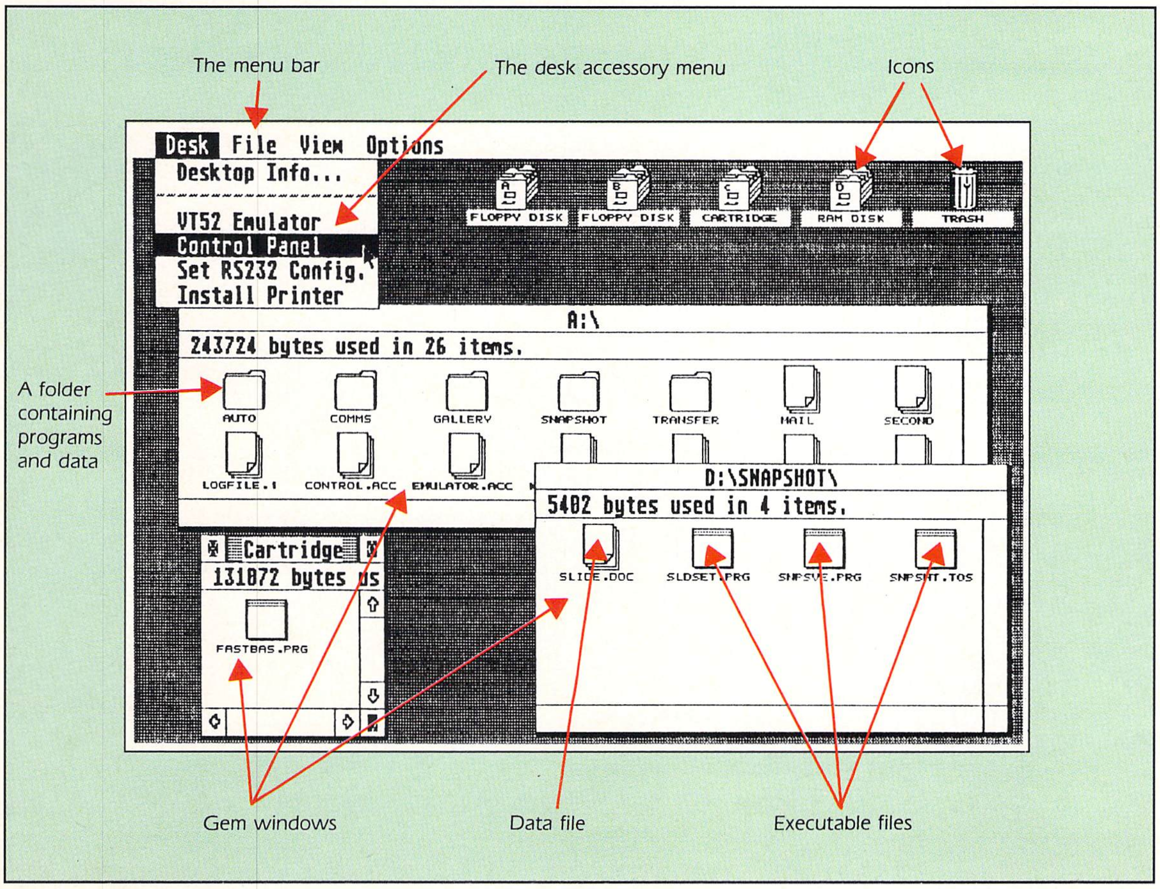
GEM has been one of the most crucial factors in the ST's success because it harnesses the sheer power of the ST, straight from the user's fingertips. In this section we'll be showing you how to access the many powerful facilities of Gem from within your own C programs.

The reason we're using C, incidently, rather than something like Basic, is simply because it's ideally suited for the job. Pascal or Modula-2 programmers will, however, have no difficulty adapting the information to their own particular requirements.

The structure of Gem

ANY wimp environment needs to perform two separate functions. Firstly, there has to be a standard method of drawing objects such as circles and boxes, and moving them around on the screen.

Gem provides you with a set of routines known as the



The Gem desktop

virtual device interface or VDI for this purpose. This supplies you with all the building blocks you need to create a wide range of impressive graphical effects.

It was originally designed to be device independent, which means you can generate graphics which will work equally well in all three of the ST's screen resolutions, or indeed on any other machine running Gem.

In theory you can also output graphics directly to other devices such as graph plotters, but unfortunately this facility has yet to be fully implemented on the ST.

The second major requirement is to allow the programmer to readily manipulate the windows, icons, menus and pointers which collectively form the heart of the wimp system. Gem contains a useful library of functions to simplify this process, called the application environment services or AES.

Gem programs

TWO distinct types of programs are supported: Applications and accessories. The former is just another name for a normal Gem program which you can execute directly from the desktop.

In contrast, accessories are loaded automatically whenever the ST is booted up. They can be accessed almost instantaneously from a special Desk menu incorporated into both the desktop and most other applications.

The ST uses a special part of the filename known as the extension to determine the nature of a program. The three characters after the dot can be one of several possibilities.

PRG or APP indicates an application, and ACC denotes an accessory. TOS or TTP indicates a program that doesn't use Gem. All other extensions such as DOC or C are treated by the ST as files containing data.

Initialisation

BEFORE you can write a Gem program you have to perform a fairly involved sequence of steps to initialise both the AES and the VDI. The procedure varies slightly depending on whether you wish to create an application or a desktop accessory.

Since applications are by far the easiest to understand, we'll temporarily restrict ourselves to these programs. The first thing you need to do is define a set of five arrays for Gem's exclusive use. These should be placed at the start of your program like:

```
int contrl[12]; /* Tells Gem which  */
                /* function to exec */
int intin[128]; /* List of integers */
                /* to be input      */
int ptsin[128]; /* List of screen   */
                /* coords input     */
int intout[128]; /* Integer results */
int ptsout[128]; /* Coordinates     */
                 /* returned        */
```

One slight snag with these arrays is that the various different C compilers define an integer inconsistently. The integers used by Gem range from –32768 to +32767, so if your C compiler uses numbers larger than these, you should precede them with the instruction *short*.

To set up the *contrl* array in Metacomco's Lattice C, for instance, you would therefore write:

```
short int contrl[12]
```

Installing the application

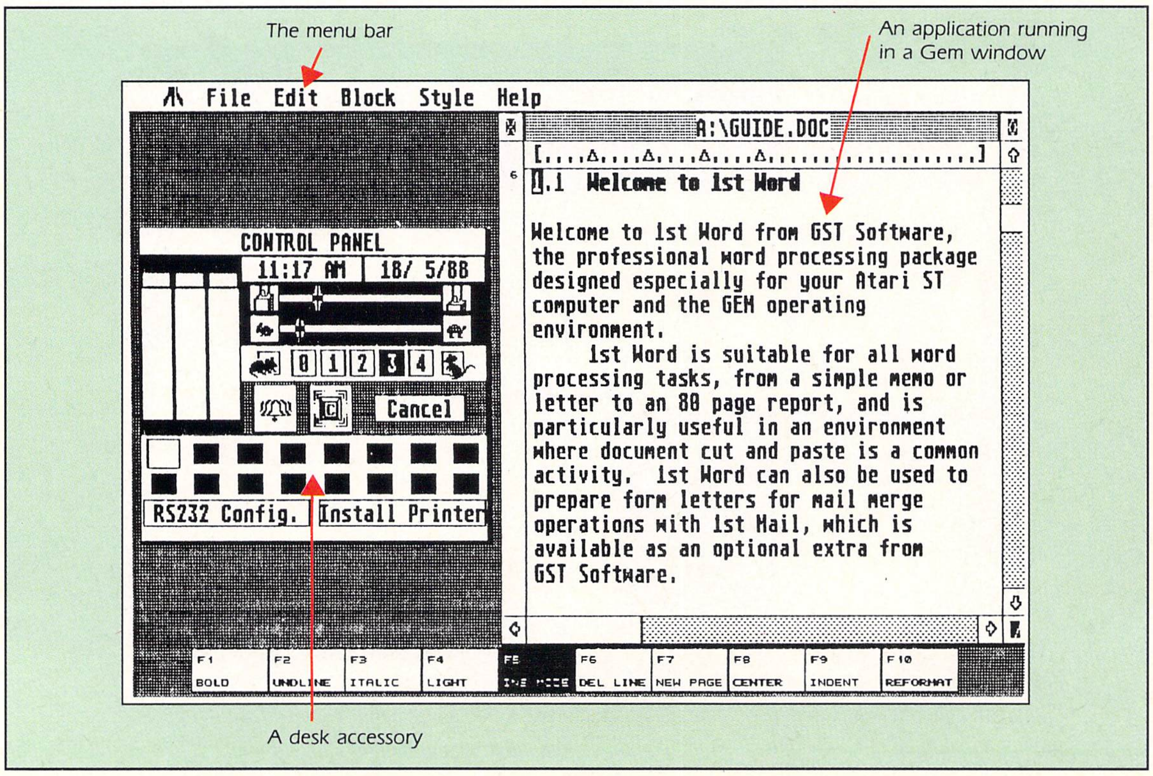
THE next step is to inform Gem you wish to install a new application into the system. This is necessary because a number of different programs can reside in memory at the same time, though only one can actually be running at once.

Since each individual program can have its own set of menus and windows, it's essential for Gem to be able to tell precisely which routine is executing. This is facilitated by the two functions *appl_init()* and *appl_exit()*, which initialise, and exit an application respectively.

Every application has a unique identification number. Generally this isn't particularly useful, but if you do need it you can use the *appl_init()* function like this:

```
ap_id=appl_init();
```

where *ap_id* is a short integer. Normally this will be positive, but in the unlikely event that it is negative, it signals that the maximum number of applications has been exceeded, and your program should be aborted.



1.1 Welcome to 1st Word

Welcome to 1st Word from GST Software, the professional word processing package designed especially for your Atari ST computer and the GEM operating environment.

1st Word is suitable for all word processing tasks, from a simple memo or letter to an 80 page report, and is particularly useful in an environment where document cut and paste is a common activity. 1st Word can also be used to prepare form letters for mail merge operations with 1st Mail, which is available as an optional extra from GST Software.

Running an application and an accessory

Handling Gem graphics

Initialising the VDI

AFTER we've installed our application we have to specify which output device we wish to use for our graphics. This may seem rather silly, as the only obvious choice would appear to be the screen, but Gem theoretically also has the ability to draw graphics directly on to plotters and laser printers.

In order to do this, it uses a function called open workstation – *v–opnwk()* – which loads a separate program known as a device driver from disc. This holds the information the ST needs to allow it to generate all the various screen effects on a particular device. Regrettably, since these extra device drivers are not yet widely available, this function is currently pretty useless.

So how do we tell Gem we wish to display our graphics on the ST's screen? Well, there's a separate instruction called *graf–handle* which returns an identification number for the screen known as the physical handle:

```
handle = graf_handle(&char_height,
&char_width,&cell_height,&cell_width);
```

The variables *char–height, char–width, cell–height* and *cell–width* simply return the height and width of each character, plus the size of the rectangular box it occupies. If this data isn't needed you can save some space by using the same variable in all four positions like this:

```
handle=graf_handle(&d,&d,&d,&d);
```

Don't forget to define the variables *handle, char–width, char–height, cell–height, cell–width,* or *d* as integers before you use them. Most Gem programs will happily work in any resolution. This is only possible because of a special function called *v–opnvwk* – open virtual screen workstation for technically minded readers.

This automatically finds out the precise dimensions of the screen, and how many colours can be displayed on it. It uses two arrays of integers, commonly called *work–in* and *work–out* which you should define at the start of the program:

```
int work_in[11];
      /* This holds a list of data   */
      /*  to be input to v_opnvwk    */

int work_out[57];
      /*  This returns the screen    */
      /*  size along with a number   */
      /*  of other useful results    */
```

A list of the usual contents of these arrays can be found in the adjoining tables. As before, these definitions should be preceeded by *short* if required by your compiler.

Although you can in practice alter the screen attributes by directly manipulating the data in the *work–in* array, the reliability of this can't be guaranteed. It's much easier to simply initialise the first 10 integers of the *work–in* array with one using a loop, and *work–in[10]* with two to specify that the ST should use normal (raster) coordinates for all graphical operations.

You can now change any of the colours, or fill types, directly using a separate set of attribute functions. The following fragment of C code demonstrates the use of this procedure.

```
for(i=0;i<10;i++)
  work_in[i]=1;
  work_in[10]=2;
  v_opnvwk(work_in,&handle,work_out);
```

This completes the Gem initialisation process.

work–in[0]	Device identifier, screen=1
work–in[1]	Line type, 1-7
work–in[2]	Line colour, 0-15, 1-3, or 0-1 depending on resolution
work–in[3]	Mark type, 1-6. Used with polymarker functions
work–in[4]	Mark colour
work–in[5]	Character fonts. 1=standard, others not included
work–in[6]	Text colour
work–in[7]	Fill type, 1=solid, 2=dotted, 3=hatched
work–in[8]	Fill index, 1-24 depending on fill type
work–in[9]	Fill colour
work–in[10]	1=NDC coordinates (32,767 x 32,767 (not implemented), 2=normal raster coordinates 640 x 400, 640 x 200, or 320 by 200 depending on resolution

Input parameters for v–opnvwk

work–out[1]	Pixel width of screen
work–out[2]	Pixel height of screen
work–out[5]	Number of text sizes
work–out[6]	Number of line types
work–out[7]	Number of line widths
work–out[8]	Number of mark types
work–out[9]	Number of mark sizes
work–out[10]	Number of character sets
work–out[11]	Number of patterns
work–out[12]	Number of hatch types
work–out[13]	Number of colours

The most useful results of v–opnvwk

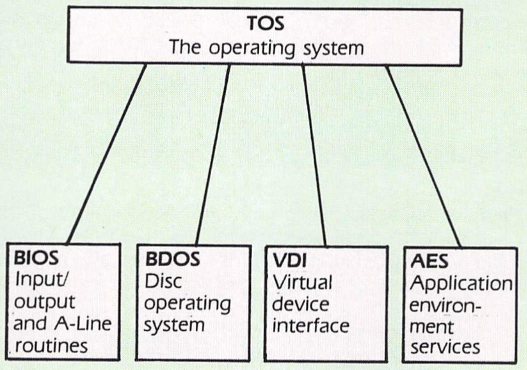
The Operating System – Tos

The Atari ST's operating is in many ways similar to CP/M 68k, and has built-in extensions to handle the mouse, Midi interface, intelligent keyboard controller and joysticks. Gem – the graphics environment manager – provides extra support for windows and graphics through four separate modules called the VDI, AES, Bdos and Bios.

The first three are machine independent and are the same on any micro supporting Gem, but the last module – the Bios – is machine specific, and contains the input/output code and primitive, but extremely fast, graphic A-Line routines for drawing pixels, lines and sprites on the screen.

The Bdos enables access to the disc drive and file management system. The AES – application environment sevices – is a multi-tasking system using a time-slicing technique. It provides a series of utilities that handle graphic based inputs such as icons, file selctors and menus.

The VDI – virtual device interface – provides a set of graphic functions that are independent of physical hardware. It enables you to define the workstation parameters governing the font and window size, define and output graphics to a device, draw lines, arcs, fill shapes, justify text.



Handling Gem graphics

Accessing the VDI

AN important side effect of *v–opnvwk* is to take the physical handle produced by *graf–handle()* and return a separate screen identifier for each application.

In the example I've stored this data in the variable *handle*. You need to use it whenever you wish to access any Gem functions. Here are a couple of typical VDI functions which illustrate this technique:

```
v_gtext(handle,x,y,"Some text...");
     /*   A standard graphic text     */
     /*   function, like Basic's      */
     /*        PRINT AT(x,y)          */

v_circle(handle,x,y,radius);
     /* This routine prints a filled */
     /* circle at x,y, with radius r */
```

If all this looks rather too complicated, don't panic, because the same code can be used again and again, whenever you create a Gem application. To make things even easier I've written two procedures called *gem–on()*, and *gem–off()* which enable you to enter and exit Gem from just a single line.

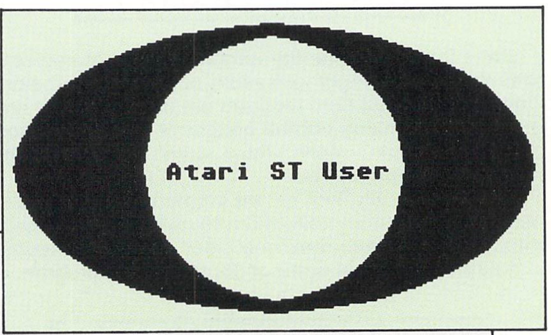
The comments are for your information only, and don't need to be typed in, and remember to define the variables *char–height, char–width, cell–height, cell–width* and the arrays *work–in* and *work–out*, before you use these routines:

```
gem_on()
{
 int I;          /* Define counter      */

 appl_init(); /* install application */

     /* Get device number for screen */
  handle = graf_handle(&char_height,
           &char_width,&cell_height,
           &cell_width);

     /*   Load work_in array with     */
            appropriate values        */
  for( I = 0; I < 10; I++ )
    work_in[I] = 1;
  work_in[10] = 2;

     /* Find screen resolution and    */
     /* return the screen identifier */
     /*  to be used in application    */
  v_opnvwk(work_in,&handle,work_out);
}
gem_off()
{
     /* Inform Gem we don't need the */
     /*  screen identifier any more   */
  v_clsvwk(handle);

     /* Tell Gem we are exiting from */
               our program           */
  appl_exit();
}
```

Try this ...

THE following listing gives you a small taste of the graphics capabilities of the VDI. We'll see shortly precisely how the functions work, and I'll show you a number of other useful VDI commands:

```
/* Simple Gem demonstration written in
Megamax C. Shows how Gem is initialised
and how VDI text and graphics functions
can be called */

#include <stdio.h>

int contrl[12];    /* Gem arrays ...    */
int intin[128];    /* short int for     */
int ptsin[128];    /* Metacomco C       */
int intout[128];
int ptsout[128];

int handle;        /* Working storage */
int char_height, char_width;
int cell_height, cell_width;
int work_in[11], work_out[57];
int x,y,radius;

main()             /* Main program      */
{
 int s;

 gem_on();         /* Initialise Gem    */
 v_clrwk(handle);  /* Call VDI to       */
                   /* clear screen      */
 x=160;            /* Experiment with   */
 y=100;            /* different x,y     */
 radius = 60;      /* and radius        */

/* Draw filled ellipse at coords x,y
with radii radius and radius*2. Fill
colour defaults to black (1)          */
 v_ellipse(handle,x,y,radius*2,radius);

/* Call an attribute function to make
   the fill colour white (0)          */
```


Atari ST User

```
 s=vsf_color(handle,0);

/* Draw a white filled circle inside */
   black ellipse                     */
 v_circle(handle,x,y,radius);

/* Print some text inside the circle */
 v_gtext(handle,x-50,y+4,"Atari ST");

 gem_off();        /* Turn off Gem      */

/* Press a key to return to desktop   */
 s = getchar();
}

gem_on()
{
  int I;

  appl_init();
  handle = graf_handle(&char_height,
           &char_width,&cell_height,
           &cell_width);
  for(I=0;I<10;I++)
    work_in[I]=1;
  work_in[10]=2;
  v_opnvwk(work_in,&handle,work_out);
}

gem_off()
{
  v_clsvwk(handle);
  appl_exit();
}
```

Filling with style

The attributes

THE short example program we looked at earlier used the mysterious *vsf_color* command to change the colour of a filled circle. We'll now go on to discuss the various attribute functions which make this possible.

The VDI supports a number of routines which allow you to determine precisely how graphics should be drawn on the screen. These range from simple things such as the ability to set the colour of lines, to the creation of complex user defined fill patterns. Let's start off by talking about the VDI set fill attribute (vsf) routines.

The VDI provides a variety of functions which enable you to select either the pattern or the colour used by fill operations. The most straightforward is *vsf_color*, which sets the fill colour to a specific value. The following examples illustrate this:

```
/* set the fill colour to white  */
s = vsf_color(handle,0);
/* draw a white circle at 10,10  */
/* with radius 15                */
v_circle(handle,10,10,15);
/* set the fill colour to black  */
s = vsf_color(handle,1);
/* draw a black ellipse at 10,10 */
v_ellipse(handle,10,10,7,15);
```

Notice that *vsf_color* returns a result in *s* indicating the colour that has actually been used by the function. Normally this will be exactly the same as the one you have designated in the instruction. But supposing you have written a program to work in one resolution, and someone tries to execute it in a different one? In this case, the colour you wish to use might not be available, and the VDI will be forced to choose another value.

In some circumstances this may be inappropriate. The information in *s* allows you to test for this eventuality, and therefore avoid any difficulty. Incidentally, when you call this function, don't forget to use the American spelling of the word colour.

Fill styles

SO far, all the examples have assumed we wish to fill our objects with one solid colour. We can however, use a number of other patterns. These can be split into four distinct groups: Solid, dotted, lined, and user.

Dotted patterns consist of an artistically arranged assortment of dots, whereas lined patterns are basically composed of straight lines. In addition there's also a special user mode which gives us the ability to define our own custom-built fill types. This mode is rather complicated and we'll consider it later.

When we initialised Gem with the *gem–on* function we effectively set the fill type to solid. If we want to change this situation we use a function called *vsf_interior*. This takes a number from one to four to determine which new fill pattern is to be selected:

1 Solid
2 Dotted
3 Lined
4 User

If we want to inform the VDI that all future fill commands should be dotted, for instance, we would use:

```
s = vsf_interior(handle,2);
```

As before, *vsf_interior* returns a value in *s*. This will be set to the fill type you have chosen – two in the above example – if the call was successful.

We now need to pick the fill style using *vsf_style()*. This gives you a choice from a wide range of different fill patterns – you can use any one of 24 dotted fills, and 12 separate lined styles:

```
/* Select fill type 4 */
s = vsf_type(handle,4);
```

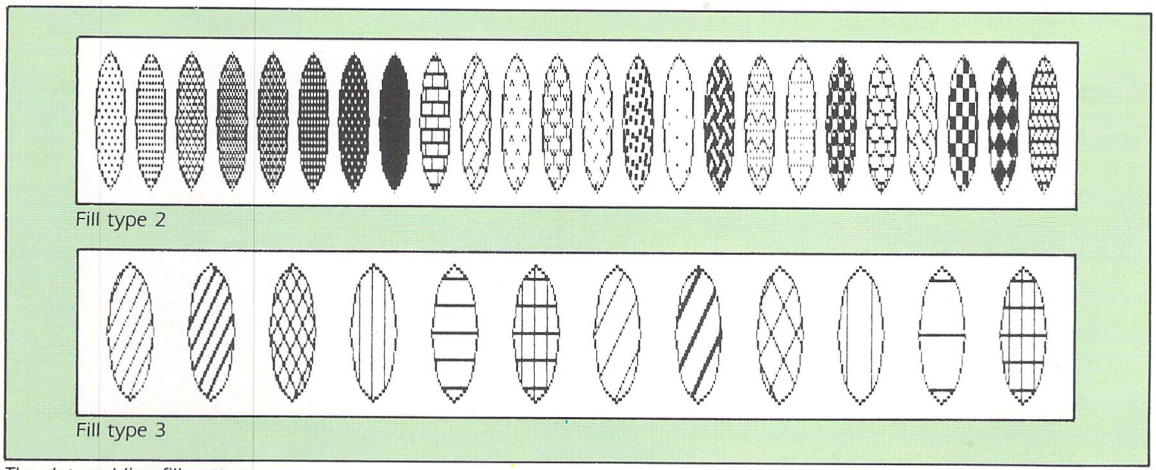
In order to set the fill attributes we therefore have to call each of these *vsf* functions in turn. Since C is an extendable language the three operations can be combined into one small C procedure:

```
fill_style(type,index,col)
int type,index,col;
{
/* s is a dummy variable, as we */
/* don't need the result        */
 int s;
/* set fill type */
 s = vsf_interior(handle,type);
/* choose style */
 s = vsf_style(handle,index);
/* select fill colour */
 s = vsf_color(handle,col);
}
```

We can simplify this routine still further with a few defines which allow you to use the actual name of the fill type when we call our function, instead of just a meaningless number, like:

```
#define solid 1
#define dotted 2
#define lined 3
#define user 4

/* Set style 12 of dotted using */
/* colour black                 */
fill_style(dotted,12,1);
```

Normally all filled objects are bounded with a line of the fill colour. This feature can be turned off using a function known as *vsf perimeter*. The action of this routine can be



The dot and line fill patterns

Filling with style

readily seen from these examples:

```
/* Switch perimeter off */
s = vsf_perimeter(handle,0);
/* Activate perimeter - default */
s = vsf_perimeter(handle,1);
```

A demonstration of this function can be found in the program below (Listing I).
To test this, remove the comments around the line:

```
/* s = vsf_perimeter(handle,0); */
```

Improving readability

ONE of the major problems encountered when writing Gem-based programs involves the complex and unfriendly names the original programmers have given to even the simplest of functions. Not only does this make our programs almost unreadable, but it also slows down the process of typing in program listings.

Fortunately we can easily get around this difficulty by making extensive use of the C preprocessor. Let's take the *v–clrwk* function used to erase the screen as an example. Although this is very easy to use, it's absolutely awful to look at. You can however, readily utilise the define instruction from C to substitute every *cls* in your program to *v_clrwk(handle)* before it is compiled like:

```
#define cls v_clrwk(handle)
```

So all you have to type in your program is *cls;* and the screen will be cleared exactly as if you had input *v_clrwk(handle).*

Another approach is to incorporate the Gem function into a new procedure with a more readable name like:

```
at(x,y,string)
int x,y;
char string[];
{
 v_gtext(handle,x,y,string);
}
```

This function can now be used in the following manner:

```
at(10,10,"Text at coords 10,10");
```

Initialising Gem

THE program below (Listing II) is a routine which displays all the different fill types on the screen at once. All the Gem initialisation code is in a special include file called GEM.H which should be saved on the disc before you try to compile the main program. Users of Megamax C should take care to ensure that GEM.H is placed in the directory HEADER.

An extra line is included in GEM.H for people with Lattice C which allows them to type in the program without modification. Note that, in order to get all the patterns on the screen at once, the program is restricted to medium or high resolution. If you want to run it in low resolution you'll therefore have to make a few minor changes to the routine.

```
*************************
   THIS IS LISTING I
*************************

/*     Show fill styles     */
/* Needs High or Medium Res */

#include <stdio.h>
#include <gem.h>

/* Working Storage */
 int coord[4];

/* Main program */
main()
{
  int i;
  gem_on(); /* Initialise GEM */
  cls;      /* Clear Screen */
/* i=vsf_perimeter(handle,0); */

  at(188,15,"List of fill patterns");

  at(280,45,"Fill Type 2");
  /* Dotted patterns  */
  for(i=1;i<25;i++) {
    fill_style(dotted,i,1);
    ellipse((i-1)*26+10,70,10,20);
  }

  at(280,135,"Fill Type 3");
  /* Lined patterns */
  for(i=1;i<13;i++) {
    fill_style(lined,i,1);
    ellipse((i-1)*52+21,160,15,20);
  }

  gem_off(); /* Leave GEM */
  at(200,192,"Press <RETURN>");
  i=getchar(); /* Press a key */
}

ellipse(x,y,w,h)
int x,y,w,h;
{
  v_ellipse(handle,x,y,w,h);
}
```

Listing I

```
***********************************
     THIS IS LISTING II
***********************************


/* GEM Include file version 1.1  */

/* Defines */
#define cls v_clrwk(handle)
#define solid 1
#define dotted 2
#define lined 3
#define user 4

/* Define for Lattice C*/
/* #define int short */

/* GEM Arrays */
int contrl[12],intin[128],ptsin[128];
int intout[128],ptsout[128];
/* Working Storage */
int handle,ch,cw,bh,bw;
int work_in[11],work_out[57];

/* GEM functions */
gem_on()
{
 int I;
 appl_init();
 handle=graf_handle(&ch,&cw,&bh,&bw);
   for(I=0;I<10;I++)
        work_in[I]=1;
   work_in[10]=2;
  v_opnvwk(work_in,&handle,work_out);
}

gem_off()
{
  v_clsvwk(handle);
  appl_exit();
}

at(x,y,string)
int x,y;
char string[];
{
v_gtext(handle,x,y,string);
}

fill_style(type,index,col)
int type,index,col;
{
 int s;
 /* set fill type   */
 s=vsf_interior(handle,type);
/* set fill style  */
 s=vsf_style(handle,index);
/* set fill colour */
 s=vsf_color(handle,col);
}
```

Listing II

Drawing polygons

We have used only a fraction of the VDI's power so far in our C programs, so we'll move on now to look at a couple of the more exciting features supported by it.

We'll begin by discussing its polyline functions. These give you the ability to draw a wide variety of many sided shapes on the screen. Each polygon can consist of anything up to 127 lines, connected together to form an enclosed surface.

The simplest example of such a polygon is the triangle, and if you wanted to plot one in Basic you would use an instruction such as:

```
LINE x1,y1 TO x2,y2 TO x3,y3 TO x1,y1
```

where *x1,y1, x2,y2, x3,y3* are the coordinates of each of the triangle's corners. Notice how the LINE statement can take a variable number of coordinate pairs. This enables you to use one function to produce anything from a line, to a complex polygon.

Unfortunately, C functions need to know in advance precisely how many parameters are to be passed every time they are executed. This means the VDI must take a rather different approach, and it stores all the coordinates of the polygon in an array. You can pass the entire array to your C routine like this:

```
v_pline(handle,n,coord);
```

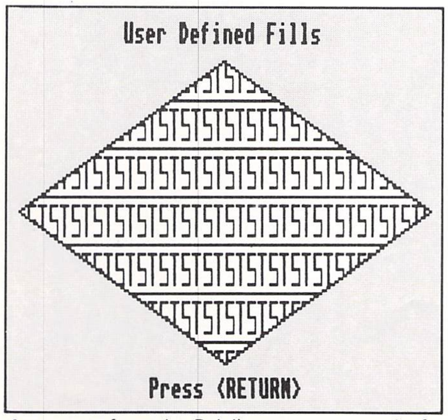
The *v_pline* function reads *n* pairs of coordinates from the array *coord*, and uses them to draw a polygon with *n*–1 corners.

The reason the number of lines is one less than the number of points is that *v_pline* draws a line from the first pair of coordinates to the second, the second pair to the third, and then successively repeats this procedure until it runs out of data. This means that in order to draw three lines on the screen you need to specify four pairs of coordinates.

Just as in the LINE statement above, if you want to create a closed polygon rather than an unconnected jumble of lines you should remember to set the last pair of coordinates to the starting point of your object.

Incidentally, the minimum number of points allowed by *v_pline* is two. You can therefore also utilise this function to draw a single line. Here is a small C procedure which uses *v_pline* to draw a triangle:

```
triangle(x1,y1,x2,y2,x3,y3)
int x1,y1,x2,y2,x3,y3;
{
/* Load in each pair of coordinates */
/* into coord array                 */
coord[0]=x1;  coord[1]=y1;
coord[2]=x2;  coord[3]=y2;
coord[4]=x3;  coord[5]=y3;
/* Set the Last coordinate pair to  */
/* the start of the triangle        */
coord[6]=x1;  coord[7]=y1;
/* Call the polyline function       */
v_pline(handle,4,coord);
}
```



An output from the Polyline program overleaf

Don't forget to define the *coord* array before calling this routine with:

```
int coord[8];
```

As a general rule the size of the coordinate array should be set to (*n*+1)*2, where *n* is the number of sides in your polygon. The *v_pline* function is fine for drawing hollow objects, but the VDI also provides a separate function to create filled polygons as well. This is called *v_fillarea*, and is used in exactly the same way as *v_pline*.

An example of *v_fillarea* can be found in the following C function which draws a filled quadrilateral on the ST's screen:

```
quad(x1,y1,x2,y2,x3,y3,x4,y4)
/* Define 4 coordinate pairs */
int x1,y1,x2,y2,x3,y3,x4,y4;
{
coord[0]=x1;coord[1]=y1;
coord[2]=x2;coord[3]=y2;
coord[4]=x3;coord[5]=y3;
coord[6]=x4;coord[7]=y4;
/* Load the start coordinates into */
/* the last part of the array      */
coord[8]=x1;coord[9]=y1;
/* Call v_fillarea with 5 sets of  */
/* coordinates                     */
v_fillarea(handle,5,coord);
}
```

Like the other VDI functions mentioned earlier, the fill type used by *v_fillarea* can be specified using the *fill_type* routine from the header file GEM.H. This procedure is also capable of installing a special user defined fill pattern. We'll now go on to explore this extremely useful facility.

Defining fill patterns

THE VDI provides the option of using either single or multicolour fill patterns. Unfortunately, the data format used by multicolour fills is rather too complex to create without the aid of a separate drawing program.

In contrast, monochrome fill patterns can be readily generated by hand with very little effort. We will therefore restrict our disussion to this type of pattern for the time being.

Note that although these patterns can only consist of two colours, they will still work perfectly well in all three graphics modes.

Before we can go any further we'll need to briefly recap number bases. If you haven't encountered this notation before, don't panic.

I'm not about to delve into any of the complexities of the subject here. All you need to know for the purposes of this discussion is that binary is a method of representing numbers using the digits zero and one instead of the more normal zero to nine.

A standard integer used by C is capable of holding numbers up to 16 binary digits long.

The reason binary numbers are so important is that they let you compact large amounts of pictorial information into a relatively small space.

One possible way this could be achieved, is to use each individual binary digit to represent either a black or a white point on the screen. A value of one in the number could indicate that the appropriate point would be black, and a zero would signify that it should be set to white. So the binary number 11110001111 might be displayed as four black pixels followed by three white then four black.

This is the storage technique used by the VDI's user defined fill patterns.

These are arranged in the form of a 16 by 16 square. Since each horizontal line can be held in a single integer, the entire fill pattern can be contained in an array of just 16 numbers.

In order to create your pattern, you should first draw your

Drawing polygons

object on a 16 by 16 square grid using a 1 for any points you want to set, and 0 for all the other points. After you have finished, you should be left with a list of binary numbers.

Unfortunately, most C compilers won't permit you to load these binary numbers directly into your array. You therefore need to convert your data into a more acceptable form before they can be used. The easiest solution to this problem is to write a small Basic program to translate the pattern into a sequence of decimal numbers:

```
 10 REM Binary to decimal converter
 20 FOR i%=1 TO 16
 30 READ a$
 40 bin%=32768
 50 byte%=0
 60 FOR j%=1 TO 16
 70 IF MID$(a$,j%,1)="1" THEN byte%=byte%+bin%
 80 bin%=bin%/2
 90 NEXT
100 IF i% MOD 4 THEN PRINT byte%;",";  ELSE PRIN
T byte%
110 NEXT
120 DATA 1000000000000001
130 DATA 0100000000000010
140 DATA 0010000000000100
150 DATA 0001000000001000
160 DATA 0000100000010000
170 DATA 0000010000100000
180 DATA 0000001001000000
190 DATA 0000000110000000
200 DATA 0000000110000000
210 DATA 0000001001000000
220 DATA 0000010000100000
230 DATA 0000100000010000
240 DATA 0001000000001000
250 DATA 0010000000000100
260 DATA 0100000000000010
270 DATA 1000000000000001
```

This prints the following list of numbers:

```
32769 , 16386 , 8196 , 4104
2064 , 1056 , 576 , 384
384 , 576 , 1056 , 2064
4104 , 8196 , 16386 , 32769
```

You can now enter this into your C program as an array definition like the following:

```
pattern[]={32769,16386,8196,4104,
2064,1056,576,384,
384,576,1056,2064,
4104,8196,16386,32769
};
```

This data can then be installed as a user defined fill pattern using the statements:

```
fill_type(user,0,1);
vsf_udpat(handle,pattern,1);
```

Since *vsf_udpat* is a rather horrible name, you can easily change it to something more readable with:

```
#define mypattern vsf_udpat
```

This allows you to replace the above call with the far friendlier:

```
mypattern(handle,pattern,1)
```

The number 1 in this instruction specifies that only one bit plane should be used. If we wanted to create multicoloured fill patterns we would change this number to either 2, (four colours) or 4 (16 colours). Our fill pattern would now need one set of 16 integers for each plane.

Every individual digit in these bit planes would then be combined to generate the larger numbers required to hold the extra colour information.

A practical example

THE following C listing shows these new functions in action. The include file GEM.H is identical with the one used on Page 6 of this section, so if you haven't typed this in, you must enter and save it.

If you look closely at the definitions for *triangle* and *quad*, you will notice that the variables *x* or *y* has been added to every coordinate before it is entered into the *coord* array. These variables are used to automatically convert the coordinates used in low resolution into the appropriate values for all three graphics modes.

```
 /* Gem polyline functions */

 #include <stdio.h>
 #include <gem.h>
 #define mypattern vsf_udpat

 int coord[16];
 int maxx,maxy,minx,miny;
 int x,y;

 /* Fill patterns */
  int stuser_pattern[]={0,7998,4104,
  4104,7944,264,264,7944,0,22391,
 21573,21621,22342,20805,30581,0};

 int st_pattern[]={0,0,32510,16400,
 16400,16400,16400,32272,528,528,528,
 528,32272,0,0,65535};

 int square_pattern[]={65535,65535,
 32769,32769,32769,32769,32769,32769,
 32769,32769,32769,32769,32769,32769,
 65535,65535};

 int pattern[]={32769,16386,8196,4104,
 2064,1056,576,384,384,576,1056,2064,
 4104,8196,16386,32769};

main()
{
 gem_on();
 cls;
/* Get size difference between low
   res & current screen */
 minx=319; miny=199;
 maxx=work_out[0]; maxy=work_out[1];
 x=(maxx-minx)/2;  y=(maxy-miny)/2;

 at(50+x,15,"The POLYLINE function");
 triangle(160,50,310,150,10,150);
 key();
 cls;

 at(50+x,15,"A filled POLYGON");
 quad(160,50,310,100,160,150,10,100);
 key();

 at(50+x,15,"User Defined Fills");
 fill_style(user,0,1);
 mypattern(handle,square_pattern,1);
 quad(160,50,310,100,160,150,10,100);
 key();

 mypattern(handle,st_pattern,1);
 quad(160,50,310,100,160,150,10,100);
 key();

 mypattern(handle,stuser_pattern,1);
 quad(160,50,310,100,160,150,10,100);

 gem_off();
 key();
}

triangle(x1,y1,x2,y2,x3,y3)
/* Draws an unfilled triangle */
int x1,y1,x2,y2,x3,y3;
{
/* Set coord array to corner points */
coord[0]=x1+x;coord[1]=y1+y;
coord[2]=x2+x;coord[3]=y2+y;
coord[4]=x3+x;coord[5]=y3+y;
/* Set last coordinate to start of
   triangle */
coord[6]=x1+x;coord[7]=y1+y;
/* Call the VDI polyline function */
v_pline(handle,4,coord);
}

quad(x1,y1,x2,y2,x3,y3,x4,y4)
/* Draws a filled four sided object
   eg. diamond */
int x1,y1,x2,y2,x3,y3,x4,y4;
{
/* Load coord array */
coord[0]=x1+x;coord[1]=y1+y;
coord[2]=x2+x;coord[3]=y2+y;
coord[4]=x3+x;coord[5]=y3+y;
coord[6]=x4+x;coord[7]=y4+y;
/* Close polygon */
coord[8]=x1+x;coord[9]=y1+y;
/* Call v_fillarea with 5 pairs of
   coordinates */
v_fillarea(handle,5,coord)+y;
}

key()
{
int c;
/* Input a key press and return */
 at(100+x,192+y,"Press <RETURN>");
 c=getchar();
}
```

A demonstration of the polyline functions

Windows on the ST

UP until now we have largely restricted our Gem tutorial to the VDI. Now we will start examining the AES – Applications Environment Services – with a simple introduction to Gem windows. These windows form the standard backdrop for the majority of Gem-based programs and can be manipulated in a number of different ways.

This interaction between the mouse and the windows is automatically managed by the AES. Unfortunately, the responsiblity of controlling the contents of these windows is left entirely with the programmer, and this is really a rather heavy burden.

You can't create a window and then simply forget about it. Whenever the size is changed or the window is moved the original contents will need to be redrawn explicitly by your program. Although you can comfortably generate a very basic window with only a few lines of code, you will have to indulge in some fairly convoluted pieces of C programming to exploit these windows. As this section progresses you will be provided with a powerful set of window management routines which should simplify this considerably.

Creating a window

THE process of window generation can be split up into three separate phases. We start off by creating our window with the *wind–create* function. This informs Gem precisely what type of window we wish to use, and installs it into the ST's memory. The syntax is:

```
whandle=wind_create(kind,tx,ty,mw,mh);
```

where *whandle* is known as the window handle which will be used for all subsequent windowing operations.

Every window we define has its own unique handle. This is only relevant to the AES window management functions, and it is not the same as the graphics handle which we obtained from *v–opnvwk()* during Gem initialisation. Normally *whandle* will be positive, but if the maximum number of allowable windows – six – is exceeded, then a negative value will be returned.

The variables *tx* and *ty* establish the coordinates of the top corner of the window when it is at its largest possible extension. Similarly, *mw* and *mh* are used to set the maximum size of our new window. Note that these values are defined in terms of the total area of the window and include the regions occupied by the Move Bar and sliders.

We specify the type of window to be installed using the variable *kind*. Each binary digit of this number fixes the status of one of 12 different window attributes – shown in the panel right. If a digit is set to one then the appropriate attribute will be included in the window definition, otherwise it will be omitted. Each separate option is converted into hexadecimal and combined with a series of logical OR operations to form the final result.

We are now able to simplify the entire process by replacing these numbers with their English equivalents using the define statement from C. Here is a simple example which should make this a little clearer. Suppose we had incorporated the following instructions at the start of our C program:

```
#define TITLE 0x001
#define INFO 0x010
```

To establish a window with a title and an information line we could now use an instruction such as:

```
handle=wind_create(TITLE|INFO,10,10,
100,100);
```

This would define a window at the top of the screen with a maximum size of 100 by 100.

Initialising

AFTER we've defined our window we must establish the initial state of the components. This can be done with the function:

```
r=wind_set(whandle,option,i1,i2,i3,i4);
```

where *whandle* is the handle of the window we installed into the system using *wind–create, option* determines the particular operation which is to be performed by this command.

Any further information which is needed by *wind–set* is normally placed in the variables *i1, i2, i3* and *i4*. Incidentally, if an error has occurred then *r* will be returned holding a value of zero.

In order to keep things simple we'll limit our discussion to just two of this command's options which enable us to set the string used by either the title or the information line of our window.

The original Gem specification required us to split this string's address between the two variables *i1* and *i2*. Fortunately most modern C compilers now allow us to input the address directly, and then automatically convert this

Defining a window

HERE is a complete list of the various possible window attributes, along with the hexadecimal numbers with which they are associated:

Name	Hex	Function
NAME	0x001	The window is to include a title in the center of the Move Bar
CLOSE	0x002	A Close Window box is to be placed at the top left hand corner of the window
FULL	0x004	A Full Window box should be inserted at the top right hand corner of the window
MOVE	0x008	The Move Bar is to be activated
INFO	0x010	An information line should be displayed just below the Move Bar
SIZE	0x020	The Size box is to be drawn at the bottom right hand corner of the window
UPARROW	0x040	The up arrow should be added to the vertical scroll bar
DNARROW	0x080	The down arrow is to be placed on the vertical scroll bar
VSLIDE	0x100	The vertical slider should be drawn on the window
LFARROW	0x200	The left arrow is to be added to the horizontal scroll bar
RTARROW	0x400	A right arrow should be placed on the horizontal scroll bar
HSLIDE	0x800	The horizontal slider should be displayed

Windows on the ST

data into the correct format during the compilation process. Look at the following example:

```
/* Option number 2 sets the */
/*    title of a window     */
/*   Megamax and Laser C    */
r=wind_set(whandle,2,name,Ø,Ø);
/*  For METACOMCO C V 3.Ø4  */
r=wind_set(whandle,2,ADDR(name),Ø,Ø);
```

The title string *name* can be any string of characters you have previously defined in your program. It can also be a literal string such as "My Window". Here are some more examples of this function:

```
r=wind_set(whandle,2,"Window title",Ø,Ø);

char title[]="Window 3";

r=wind_set(whandle,2,title,Ø,Ø);
```

Similarly, we can also install the information line in the same way using an *option* number of three:

```
r=wind_set(whandle,3,
"Information line",Ø,Ø);
```

It is important to note that the *wind–set* command loads the address of the string rather than its contents. If this string is subsequently modified the title or information line will be changed whenever the appropriate window is redrawn by the AES.

Once we have created a window and designated its various attributes, we can finally display it on the ST's screen using the AES function *wind–open*:

```
r=wind_open(whandle,x,y,w,h);
```

where *whandle* indicates which of the windows currently installed in memory is to be displayed. The variables *x, y, w* and *h* determine the current dimensions. These can be anything up to the maximum size we specified using our initial call to *wind–create*.

Now for a small example:

```
/* Display a window created by
   wind_create at 10,10 with
   dimensions 20 x 20          */
r=wind_open(whandle,10,10,20,20);
```

Note that *wind–open* does not erase the contents of the window. This can occasionally lead to some rather unusual effects, and it's normally a good idea to clear this area after it has been drawn.

We are now in a position to summarise the three phases involved in the creation of a Gem window:

- Install the window into the system with *wind–create*.
- Set each of the attributes in turn using *wind–set*.
- Display it with *wind–open*.

A demonstration of this procedure can be found in the *windopen* function included in the listing below.

Try this...

THE example program is split into two distinct parts which should be entered seperately. The first section is a slightly ammended version of the standard header file GEM.H. The program itself simply displays a window on the screen and then waits for you to press a mouse button before returning to the desktop.

Program I

```
/* GEM Include file - MEGAMAX C */

/* GEM Arrays */
/* Defines    */

#define NAME Øx0Ø1

/* Define for Lattice C */
/* #define int short */
/* #define ADDR(a) ((long)a)>>16,((lo
ng)a)&Øxffff */

/* GEM Arrays */
 int contrl[12];
 int intin[128];
 int ptsin[128];
 int intout[128];
 int ptsout[128];

/* Working Storage */
 int handle,char_height,char_width;
 int cell_height,cell_width;
 int work_in[11],work_out[57];

gem_on()
{
/* Define loop counter I */
   int I;
/* Tell GEM we wish to install
    another application */
   appl_init();
/* Get Device Number for screen */
   handle=graf_handle(&char_height,
&char_width,&cell_height,&cell_width);
/* Load work_in array with
   appropriate values */
   for(I=Ø;I<1Ø;I++)
       work_in[I]=1;
   work_in[1Ø]=2;
/* Find out screen res and return
   the screen identifier to be */
/* used in this application */
   v_opnvwk(work_in,&handle,work_out);
}

gem_off()
{
/* Inform GEM that we don't need the
   screen identifier any more */
  v_clsvwk(handle);
/* Tell GEM we are exiting from our
   program */
  appl_exit();
}

at(x,y,string)
/* define x,y coordinates */
 int x,y;
/* string is a pointer to a string
   of characters */
  char *string;
{
/* Call VDI */
v_gtext(handle,x,y,string);
}
int click()
{
  int button,x,y;
  button=Ø;
  while (button==Ø) vq_mouse(handle,
&button,&x,&y);
  return(button);
}
```

Program II

```
/* A Gem simple window  */

#include <stdio.h>
#include <d:\gem.h>

int windo[8],wx[8],wy[8],ww[8],wh[8];
int x,y,h,w;
int s;
main()
{
   gem_on();
   wx[1]=1Ø;wy[1]=16;
   ww[1]=3ØØ;wh[1]=2ØØ;
   windo[1]=windopen(1,"Gem Window");
   s=click();
   windclose(1);
   gem_off();
}

windopen(n,title)
int n;
char *title;
{
  int wr,whandle;
/* Create a window */
  whandle=wind_create(NAME,wx[n],
wy[n],ww[n],wh[n]);
/* Test for success */
  if (whandle<Ø) return (Ø);
/* Set title using wind_set opt 2 */
  wr=wind_set(whandle,2,title,Ø,Ø);
/* Open window */
  wr=wind_open(whandle,wx[n],wy[n],
ww[n],wh[n]);
/* Return window handle */
  return (whandle);

}

windclose(w)
int w;
{
   int wr,h;
/* Get handle from global array */
   h=windo[w];
/* Close window */
   wr=wind_close(h);
/* Delete window from memory */
   wr=wind_delete(h);
}
```

Wiping windows

WHEN we have finished with our window we will ultimately wish to wipe it from the ST's memory. The AES provides us with two useful functions for this purpose – *wind_close* erases the window from the screen. It can be subsequently redisplayed at any time using a single call to the *wind_open* function.

Another important command is *wind_delete* which deletes a window definition completely. One side effect is to release the old window handle for future use. You should always remember to call *wind_close* before using *wind_delete*. The format of these two instructions is identical:

```
r=wind_close(whandle)
r=wind_delete(whandle)
```

In both cases *r* will be set to zero if an error has occurred.

Clipping output

A RATHER better idea is to assume that every window contains just the visible portion of a larger "virtual" screen. You can implement this approach by simply clipping away the parts of your image which lie outside the current window before they are drawn. The VDI provides a useful *vs_clip* command for this purpose which limits the action of the VDI's drawing operations to a specific region of the screen. The format is:

```
vs_clip(handle,flag,coord);
```

The *handle* parameter is the standard screen handle used by the VDI. Don't confuse this with the window handle returned by the *wind_create* function. The *flag* parameter is used to control the action of the *vs_clip* function. A value of one at this position turns the clipping on and a value of zero restores the VDI's drawing operations to normal.

The final parameter, *coord*, is a list of integers used to determine the position and size of the clipping rectangle. These numbers hold the coordinates of the interior of our window.

coord[0]	= Top left x coordinate
coord[1]	= Top left y coordinate
coord[2]	= Bottom right x coordinate
coord[3]	= Bottom right y coordinate

Look at the following procedure. This calls the *vs_clip* function to limit the VDI's drawing operations to a particular window.

It assumes, of course, that the current dimensions of the window have been previously placed in the arrays *wx, wy, ww* and *wh*.

```
wclip(w)
int w;
{   int coord[4];
    coord[0]=wx[w];
    coord[1]=wy[w];
    coord[2]=wx[w]+ww[w];
    coord[3]=wy[w]+wh[w];
    vs_clip(handle,1,coord);
}
```

In order to demonstrate how this clipping works take a look at Figures I and II which show the result of drawing a filled circle inside a simple Gem window. Figure I is the unclipped version, and as you can see, the circle completely overwrites the window's borders.

If however, the drawing is clipped using *vs_clip*, only the parts of the circle which lie inside the window will be drawn. This can be readily observed from Figure II.

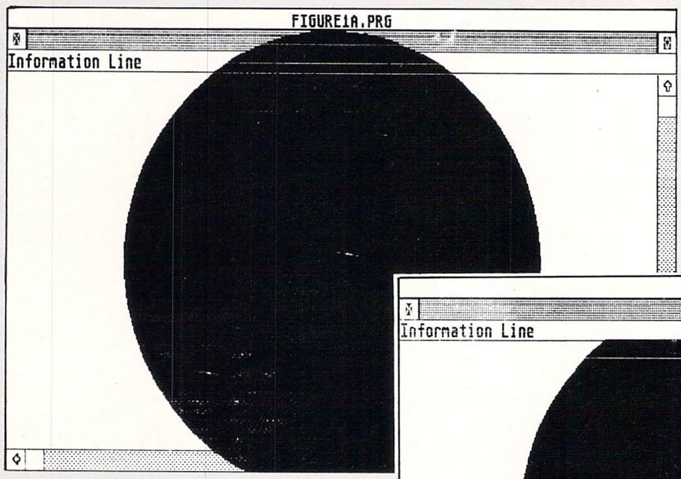


Figure I: The unclipped circle overwrites the window border

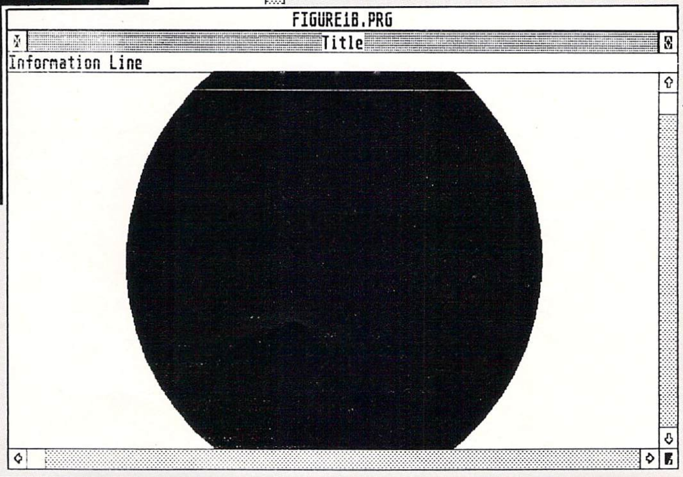


Figure II: After setting the clipping rectangle the top and bottom of the circle is clipped to fit the window

Creating windows

Scaling images

AS we have stressed, the overall responsibility for the management of a window's contents rests entirely on your shoulders. This task is complicated by the fact that the VDI doesn't completely understand about Gem windows. You are therefore forced to generate all your graphics using absolute screen coordinates. So you can't simply draw something inside a particular window, irrespective of its size and position on the ST's screen.

There are two possible solutions to this problem. First, you can write a special routine to scale your image to the correct size before you draw it. This effectively treats each window as just a miniature version of the entire screen.

Unfortunately, this technique is only really useful when you are displaying an actual picture of some sort. If you attempted to output a piece of text to such a window the contents would rapidly become totally unreadable.

Calculating window sizes

IT is important to remember that each window is effectively defined by two different sets of coordinates. One holds the total dimensions of the window, and includes the regions occupied by the title line and the window borders. These are the coordinates which are required by the *wind_create* and *wind_open* functions during window initialisation.

Additionally, there's also a second set which represents the inner working area of this window. It is these coordinates which are used in conjunction with the *vs_clip* command. When we are manipulating a window it's often useful to be able to convert between the working coordinates and the total coordinates. The AES incorporates a helpful function called *wind_calc* which performs this operation for us directly:

```
r=wind_calc(flag,type,wxin,wyin,wwin,
    whin,&wxout,&wyout,&wwout,&whout);
```

The parameter *flag* determines which of two possible operations are to be performed by this function. A value of zero indicates that the coordinates of the working area of the window should be converted into the total area. Alternatively, you can also calculate the working area of the window from the total area by setting the flag to one.

The next parameter, *type*, consists of a bit field representing the window attributes you have selected. The format is identical to the equivalent parameter used by the *wind_open* and *wind_create* functions.

Before the *wind_calc* function can be called, *wxin, wyin, wwin* and *whin* must be loaded with the appropriate input coordinates. These may hold either the working area or the total area, depending on the precise operation you have selected.

Once the calculation has been performed, the resulting set of coordinates will be placed in the variables *wxout* to *whout*. If an error occurred during the function, a value of zero will be returned in *r*. Any other number indicates that the operation was a success.

At first glance *wind_calc* may look extremely complicated, but in practice it's surprisingly easy to use. Here are a couple of simple examples which should make things a little clearer. Supposing *wx,wy,ww* and *wh* currently hold the total coordinates of a window. You could convert these into the coordinates of the working area using a line like:

```
r=wind_calc(1,TITLE,wx,wy,ww,wh,&wx,
    &wy,&ww,&wh);
```

The variables will now contain the working coordinates of the window. These could then be used in a subsequent call to *vs_clip*. Similarly, you could convert these new coordinates back to the total area like so:

```
r=wind_calc(0,TITLE,wx,wy,ww,wh,&wx,
    &wy,&ww,&wh);
```

Now for a somewhat larger example.

```
/* Define clip array */
int cp[4];
/* define working variables */
int wx,wy,ww,wh,r,whandle;
/* Set the coords of total area */
wx=50;
wy=50;
ww=100;
wh=100;
/* Create window using coords */
whandle=wind_create(NAME,wx,wy,ww,wh);
/* Open the window */
r=wind_open(whandle,wx,wy,ww,wh);
/* Set name of window */
r=wind_set(whandle,2,"Title");
/* Get coordinates of working area and
   place them into clip array */
r=wind_calc(1,NAME,wx,wy,ww,wh,
   &cp[0],&cp[1],&cp[2],&cp[3]);
/* Activate clipping */
vs_clip(handle,1,cp);
/* Draw circle inside window */
v_circle(handle,150,150,100);
```

Note that the above routine will not compile as it currently stands, as it omits a large chunk of essential Gem initialisation. If you're feeling adventurous however, you can easily expand this fragment into a real program.

Limitations

ONE serious limitation of the *wind_calc* function is that it assumes that you already know one of the sets of coordinates in advance. Unfortunately, this information is often simply not available. In these circumstances, you can utilise the AES command *wind_get*. This is a powerful function which can return an impressive range of information about any window which has previously been defined using *wind_create*.

The format of the wind_get command is:

```
r=wind_get(whandle,option,&i1,&i2,
    &i3,&i4);
```

where *whandle* is the handle of the window you wish to examine, *option* indicates one of a number of alternative operations which can be executed by this command, and *i1, i2, i3*, and *i4* are a set of variables which will be used to hold the results. As you might expect, their exact significance varies considerably, depending upon the particular operation you have chosen. The variable *r* indicates whether an error has occurred during the command. If a value of zero is returned, then the call to *wind_get* has failed in some way.

For the moment we'll restrict ourselves to the two commands which are directly relevant to this present discussion. These are options number four and five, which return either the working coordinates, or the total coordinates of a window, respectively.

After these functions have terminated, the appropriate coordinates will be placed in the variables *i1-i4* like so:

```
i1 = x coord of top corner
i2 = y coord of window
i3 = Width of window
i4 = Height of window
```

The standard use of this command is to determine the maximum area available from the Gem desktop. This enables you to position your window so that it doesn't accidentally overwrite the Gem menu line. The desktop is automatically assigned a window handle of zero, so if you wanted to calculate its working area you could call *wind_get* in the following manner.

```
r=wind_get(0,4,&wx,&wy,&ww,&wh);
```

You can also use this function to work out the area of the total screen.

Putting theory into practice

Multiple windows

WE have learnt a fair bit of theory about Gem windows, so now let's put that theory into practice with an example listing. What we'll do is to define and open five overlapping windows on the ST's screen. A filled circle is then drawn inside each one and is clipped using the *wclip* function. You can then successively remove each window from the screen by clicking on the left mouse button.

The program below has been designed to run in monochrome, and if run in other resolutions some of the windows may be off the screen. Also the GEM.H include file from page 10 in this section is required in order to define some functions used by the program.

Changing a window's size

ONCE a window has been successfully installed into the ST's memory its size and position can easily be changed using an option from the *wind_set* function. We first encountered this function during the window initialisation process, when it was used to set the title and information line of a window. As before, the format of the *wind_set* function is:

```
r=wind_set(whandle,option,I1,I2,I3,I4)
```

where *r* is a number returned by *wind_set*. A value of zero indicates that an error of some sort has occurred.

The *whandle* parameter is the handle of the window we want to change and *option* is the number of the function we wish to call. In this case, we will be using option number five.

The variables *I1* and *I2* hold the new screen coordinates of the window and *I3* and *I4* contain its new size. The call to change a window's size can be performed with:

```
r=wind_set(whandle,5,x,y,w,h);
```

Note that *x, y, w* and *h* refer to the total coordinates of the new window. If the working coordinates are required, they will need to be calculated in the normal way using the *wind_calc* function.

Window manipulation

THE AES automatically takes care of many of the more arduous aspects of window management. Whenever you click on the size box of a window a resizable outline is displayed on the ST's screen. Similarly, clicking on the move bar generates a hollow box which can be dragged around to change the window's position. These movements are controlled directly from the AES.

It is important to realise, however, that the results of these actions need to be dealt with explicitly by your own program.

The AES only informs you that the user has requested an action – it does not perform that action itself. So if a window is moved or resized its dimensions will need to be changed using an immediate call to the *wind_set* function in your program.

Events

A SPECIAL mechanism is provided by the AES to allow you to control the various parts of the Gem system. It takes the form of a series of commands which wait for a specific event to occur. Most of these events are generated

```
/* Multiple Windows  */

/* Include Files */

#include <stdio.h>
#include <gem.h>

/* Global Arrays */

/* Storage for window coordinates */
int windo[8],wx[8],wy[8],ww[8],wh[8];
/* Storage for window titles */
char wtitle[8][12];

/* Variables */
int x,y,h,w,i,j,s;

main()
{
/* Initialize GEM */
  gem_on();

/* Open 5 windows */
for (i=0;i<5;i++)
{
 /* Set position of window */
 wx[i]=10+i*20;    wy[i]=40+i*40;
 ww[i]=200;        wh[i]=150;
 /* Initialize window title */
 sprintf(wtitle[i]," Window %d",i);
 /* Open a window */
 windo[i]=windopen(i,wtitle[i]);
 /* Draw filled clipped circle */
 redraw(i);
}

/* Close 5 windows */
for (i=0; i<5; i++)
{
/* Read Mouse buttons */
s=click();
windclose(windo[i]);
}
/* Kill GEM */
   gem_off();
}


/* Function to open a window */
int windopen(n,title)
int n;
char *title;
{
 int wr,whandle;
 int xs,ys,ws,hs;
/* Calculate exterior size of the
   window specified by x,y,w,h */
 wr=wind_calc(0,NAME,wx[n],wy[n],
ww[n],wh[n],&xs,&ys,&ws,&hs);
/* Create a window with a title */
whandle=wind_create(NAME,xs,ys,ws,hs);
 if (whandle<0) return(0);
/* Set window name to title */
 wr=wind_set(whandle,2,title,0,0);
/* Open window */
 wr=wind_open(whandle,xs,ys,ws,hs);
return(whandle);
}

/* Function to close a window */
windclose(w)
int w;
{
 int wr;
 if (w>0)
 {
 wr=wind_close(w);
 wr=wind_delete(w);
 }
}

/* Function to clear a window */
clearw(n)
int n;
{
int coord[4];
coord[0]=wx[n];
coord[1]=wy[n];
coord[2]=wx[n]+ww[n]-1;
coord[3]=wy[n]+wh[n]-1;
vsf_color(handle,0);
vr_recfl(handle,coord);
}

/* Function to clip a window */
wclip(w)
int w;
{
int coord[4];
coord[0]=wx[w];
coord[1]=wy[w];
coord[2]=wx[w]+ww[w]-1;
coord[3]=wy[w]+wh[w]-1;
vs_clip(handle,1,coord);
}

/* Redraw a window's contents */
redraw(w)
int w;
{
wclip(w);  /* Clip window */
clearw(w); /* Clear window */
/* Draw circle */
vsf_color(handle,1);
v_circle(handle,wx[w]+ww[w]/2,
wy[w]+wh[w]/2,100);
}
```

Putting theory into practice

by an input of some sort.

Typical events include:

● *Keyboard event* – a character has been typed in at the keyboard.

● *Time event* – the AES waits until a certain amount of time has elapsed.

● *Mouse event* – the mouse has been moved into or out of a rectangular area. This feature is occasionally used to construct customised dialogue boxes, such as HiSoft's file selector.

● *Mouse button event* – one or both of the ST's mouse buttons has been clicked.

● *Message event* – the user has manipulated a window or accessed one of the Gem menus.

● *Multiple event* – checks for a whole list of events. This can comprise of any of those above.

The AES incorporates separate functions for each of the possible events. So if you wanted to ask the AES to wait for a keyboard event, you could call the *evnt_keybd* function like this:

```
key=evnt_keybd();
```

where *key* is just the Ascii code of the key which has been pressed.

We'll now look at the *evnt_mesag* routine which waits for the manipulation of a Gem window. The format is:

```
evnt_mesag(message);
```

Here *message* is an array of eight integers which will contain the results of the function. This array must have been previously defined in your program with a line like:

```
int message[8];
```

The *evnt_mesag* function returns one of a number of messages which indicate the specific event which has been detected. The nature of the message can be found in *message[0]*.

Here is a list of some of the events which can be generated by a Gem window, along with the expected results:

Message number 21: WM_TOPPED

A topped message is produced when a new window is selected with the mouse and *message[3]* contains the handle of the new window to be activated. The *wind_set* command provides a special function to activate this:

```
wind_set(whandle,10,0,0,0,0)
```

where whandle is the handle of the window which has been chosen.

Message number 22: WM_CLOSED

This code is returned whenever the mouse is clicked on the Close box of a window and *message[3]* holds the handle of the window to be closed. Your program should handle this event by closing the window using *wind_close* and possibly deleting it from memory with *wind_delete*.

Message number 23: WM_FULLED

This message is generated when the mouse is clicked on the Full box of the window and *message[3]* contains the handle of the window to be fulled. This event can be performed by changing the size of the window to the maximum possible dimensions using the *wind_set* command.

Message number 27: WM_SIZED

The size box has been used to request a new size for the current window. This must be accomplished directly from your program with a call to *wind_set*.

Return values: *message[3]* contains the handle of the window to be resized, *message[4]* is the current X coordinate of the window, *message[5]* is the current Y coordinate, *message[6]* is the new width of the window and *message[7]* is the new height of the window.

Message number 28: WM_MOVED

The move bar has been dragged to reposition one of the windows. This needs to be implemented using an explicit call to *wind_set* in your program.

Return values: *message[3]* contains the handle of the window to be moved, *message[4]* is the new X coordinate of the window, *message[5]* is the new Y coordinate, *message[6]* is the current width of the window and *message[7]* is the current height.

The standard way of dealing with these messages is to place all the code into a simple loop. This can be summarised by the following steps:

● Wait for a message event (call *evnt_mesag*)

● Check for each of the possible window events. The precise events which need to be tested will vary depending on your current window definition. These tests are normally implemented using the *switch* statement.

● If all the windows have been closed jump back to your main program, otherwise the first two steps should be repeated.

Window Control

THE basic principles of controlling a Gem window are best demonstrated with an example. Look at the following *window_control()* routine:

```
window_control()
{
/* Assumes that:
 full = global integer variable.
 message = global array of 8 integers.
 Calls the functions redraw,
 window_move, and window_full.
 These need to be defined separately.
*/

/* Define Storage */
int open,whandle,option;
/* Continue until window closed */
open=1;
/* Set flag denoting the status
   of current Window */
full=0;
/* Wait until something interesting
   happens */
 do
 {
 evnt_mesag(message);
 option=message[0];
 whandle=message[1];
 switch(option)
 {
 case WM_TOPPED:
 wind_set(whandle,10,0,0,0,0);
 break;

 case WM_FULLED:
 window_full(whandle);
 redraw(whandle);
 break;

 case WM_SIZED:
 case WM_MOVED:
 window_move(whandle);
 redraw(whandle);
 break;

 case WM_CLOSED:
 open=0;
 break;
 }
 }
 while (open);
}
```

Note that this function is only intended as an illustration, and should not be run in isolation. It includes calls to a number of window management functions which we will be looking at next.

Manipulating windows

Redrawing a Gem window

IT is often necessary to redraw a Gem window's contents from time to time. For instance, when it is resized, moved, topped or overwritten its contents may be corrupted. Here is a simple C function which allows you to redraw a window's contents after it has been moved or resized:

```
  /* Redraw a window's contents      */
redraw(w)
  /* w=current window handle. Expects
     window coordinates to be stored
     in the arrays wx,wy,ww,wh       */
int w;
{
int ew,eh;
v_hide_c(handle,0);
wclip(w);   /* Clip window */
clearw(w);  /* Clear window */
  /* Draw circle */
vsf_color(handle,1);
ew = wx[w]+ww[w]/2;
eh = wy[w]+wh[w]/2;
v_ellipse(handle,ew,eh,ww[w]/2,
          wh[w]/2);
v_show_c(handle,0);
}
```

The function *redraw(w)* uses the *wclip* we developed earlier to recreate the contents of the current window at its new position.

Note that *redraw(w)* currently assumes that there is only a single window on the screen. If we wish to use multiple windows this function will have to be expanded considerably.

Fulling a window

ONE of the common requirements of a window management routine is the ability to enlarge a window to fill the whole screen. This can be performed using the procedure *window_full(w)*:

```
window_full(w)
  /* Assumes that the window coords
     are held in arrays wx,wy,ww,wh.
     w is the handle of the current
     window.
     full is a global variable       */
{
int xs,ys,ws,hs;
if(full)
{
  /* Do nothing */
full=0;
return;
}
else
 {
  /* Get working area of desktop     */
wind_get(0,4,&xs,&ys,&ws,&hs);
  /* Set fulled flag to 1            */
full=1;
  /* Set new window size */
wind_set(w,5,xs,ys,ws,hs);
  /* Get working area of new window */
wind_get(w,4,&wx[w],&wy[w],&ww[w],
         &wh[w]);
}
}
```

'he first action of this procedure is to check whether the ıdow is already full. If so, control is returned back to the function which called it. Otherwise, the current window is set to the maximum area available from the desktop using the *wind_set* command. If you intend to use *window_full(w)* within your own programs you will have to set the variable *full* to zero before calling it.

Moving a window

WHEN we created the *window_control* function we needed a way of moving and resizing a window on the screen. The standard method of achieving this effect requires the use of the AES function *wind_set*. This can be seen in the following C procedure:

```
window_move(w)
  /* Expects window coordinates to be
     stored in arrays wx,wy,ww,wh and
     w is the handle of the current
     window. The message array must
     have been defined previously. It
     is assumed to have been returned
     by the evnt_messag function     */
int w;
{
  /* Define coordinates */
int xs,ys,ws,hs;
  /* Get new coordinates */
xs=message[4];ys=message[5];
  /* Get new size */
ws=message[6];hs=message[7];
  /* Check for minimum width */
if (ws<40) ws=40;
  /* Check for minimum height */
if (hs<50) hs=50;
  /* Move window */
wind_set(w,5,xs,ys,ws,hs);
  /* Get new working coordinates */
wind_get(w,4,&wx[w],&wy[w],&ww[w],
         &wh[w]);
}
```

The effect of *window_move* is to set the new dimensions of the window with *wind_get* and then calculate the appropriate working coordinates. See how we've checked for the minimum width and height. This prevents you from selecting a window which is too small to hold a reasonable amount of information.

The function *window_move* was designed specifically for use with the AES *evnt_mesag* routine. If you want to control a window independently of this function you will have to change the lines which get the new dimensions of the window. That is:

```
xs = message[4];
ys = message[5];
ws = message[6];
hs = message[7];
```

The variables *xs* and *ys* should be loaded with the new screen coordinates of the window and *ws* and *hs* with the required dimensions.

If you're familiar with the Gem windowing system you will have already encountered the vertical and horizontal scroll bars used to control a window's contents. We'll now discuss how we can use these features from one of our own programs.

The first requirement is to incorporate the new attributes into the existing window definition. This can be done by changing the property list used in the original call to the *wind_create* function. If, for instance, we wanted to install a window at the centre of the screen, we could execute the lines:

```
kind=NAME|CLOSE|MOVE|SIZE|VSLIDE|HS
LIDE;
whandle = wind_create(kind,200,100,
100,50);
```

Manipulating windows

A public domain C compiler

IF you'd like to be able to follow this Gem programming section, but are unwilling to spend more than £100 on a commercial C compiler, you might be interested in a version of this language which has recently appeared in the public domain. It is called Sozobon C and can be bought for around £3.50 from your local PD library.

Sozobon C appears to conform to the complete Kernighan and Ritchie specification. It includes a full set of Gem libraries and is easily capable of compiling all the example programs which have been featured in this Gem tutorial.

As you might expect from a PD program, there are a couple minor limitations to the system.

Also note that the linker supplied occasionally generates a harmless error message during compilation of some windowing routines. Fortunately this error is NOT fatal, and has no effect whatsoever on the resulting programs.

Sozobon C is currently supplied on one double sided 720k disc, but as the package also works perfectly on an unexpanded 520ST, it's well worth shopping around for a copy on single sided discs.

Sozobon C is one of those rare programs which deserves a place in everyone's library. So get hold of a copy without delay – and C for yourself!

Additionally, we would also have to add the amended attribute list to any subsequent calls to *wind_calc*. Once we have initialised the sliders we then need some way of manipulating them from within the programs.

The AES includes a range of commands which allow us to change either the size or the position of the two scroll bars. These can be accessed through the *wind_set* and *wind_get* functions. The formats of these commands are:

```
r = wind_set(whandle,option,setting);
    /* Set a slider */
```

and:

```
r = wind_get(whandle,option,&setting);
    /* Read a slider */
```

The *option* parameter holds the appropriate function number and *setting* contains a value from 0 to 1,000. This represents either the new or the current status of the slider, depending on whether we have called the *wind_set* or the *wind_get* command. As usual, *r* holds the result of the function. If an error has occurred, a zero will be returned in this position.

Here is a full list of the available commands:

Option 8: Sets/reads the current position of the horizontal slider. This can range between 0 (far left) and 1,000 (far right).

It is important to realise that the value is only relative to the current window. So if its size is subsequently changed, the physical appearance of the slider on the ST's screen will be adjusted automatically.

Also, as the slider's precise dimensions will vary depending on the size of the window there is no guarantee that any individual movement will actually be reflected on the ST's screen:

```
    /* Moves the slider to the
       middle of the window     */
wind_set(whandle,8,500);
```

Option 9: Sets/reads the position of the vertical slider. A setting of 0 represents the top of the window and a value of 1,000 the bottom:

```
    /* Reads the current position of the
       vertical slider and places it in
       the variable pos                 */
wind_get(whandle,8,&pos);
```

Option 15: Sets/reads the size of a horizontal slider bar. This can range in size from a small square (–1) to the maximum width of the current window (1,000):

```
    /* Sets slider to half the
          available space       */
wind_set(whandle,8,500);
    /* Reads slider into the
          variable size     */
wind_get(whandle,8,&size);
```

Option 16: Sets/reads the size of a vertical slider. The height of the slider can be set to any value from –1 (very small) to 1,000 (the total extent of the current window).

Controlling sliders

THE AES windowing system allows the window sliders to be moved with just a simple click of the mouse pointer. This is controlled using the event handler we encountered in the section on *evnt_mesag*.

We'll now examine the different messages which can be generated by these sliders.

Message 25: WM_HSLIDE

This is produced when the horizontal slider bar in the current window is manipulated. Once the change has been detected the program should immediately update the position of the slider using a call to the *wind_set* function:

```
wind_set(whandle,8,pos);
```

Where *pos* holds the new position of the horizontal slider. The return values are:

```
message[3]: Contains window handle
message[4]: New position of slider
```

Message 26: WM_VSLID

This is generated if the vertical slider is moved. As before, the new slider position should be permanently set using the appropriate option from *wind_set*. The return values are:

```
message[3]: Contains window handle
message[4]: New slider position
```

We could now incorporate the code to control these sliders directly into the *case* statements used in our previous *gem_control* function like:

```
case WM_HSLIDE:
     xslide=event[4];
     wind_set(w,8,xslide);
     redraw(w);
     break;

case WM_VSLIDE:
     yslide=event[4];
     wind_set(w,9,yslide);
     redraw(w);
     break;
```

We would also have to modify the *redraw* function to display the relevant sections of our windows, depending on the values stored in the variables *xslide* and *yslide*.

How to program in C

ALONG with 68000 machine code, the programming language C is probably the most important language to learn. The vast majority of serious software – word processors, text editors, spreadsheets, databases, utilities and so on – are written in C. Even the Gem operating system is largely a C program.

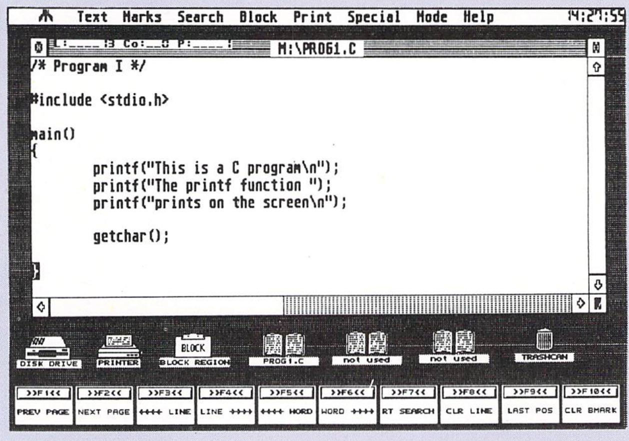
So if you want to write this type of software package, or even if you simply fancy investigating the latest fad in computer programming languages, then C is a must.

History

THE roots of C can be found in CPL – Combined Programming Language – created by the University Mathematical Laboratory in Cambridge and the University of London Computer Unit.

However, the language was far too bulky for most applications programmers had in mind, so Martin Richards at Cambridge developed a cut down version called BCPL – Basic CPL. This was condensed even further by Ken Thompson at Bell Laboratories in New Jersey and he created B.

Dennis Ritchie, one of his colleagues with whom he was working to create a new operating system called Unix, further developed B and it became the language C. Unix was then almost entirely rewritten in C, which is why it is so closely associated with it.



```
/* Program I */

#include <stdio.h>

main()
{
        printf("This is a C program\n");
        printf("The printf function ");
        printf("prints on the screen\n");

        getchar();

}
```

Creating the C source code using a text editor

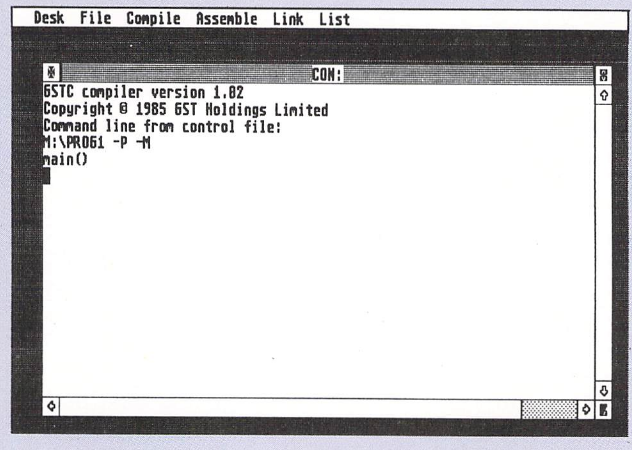
Programming in C

C PROGRAMS are written using a text editor which produces Ascii files. The source code – the C program listing – is then taken by a compiler and turned into a machine code program which can be run regardless of whether the compiler is present or not.

The C language itself is very small, and the vast majority of functions and procedures found in other languages are missing. They are provided separately and are stored in files on disc called libraries.

The compiler can be instructed to include the contents of these libraries in the program at compile time. A C compiler is often judged by the size and quality of the ready-made libraries provided.

Creating a C program is a bit like putting a jigsaw puzzle together – it's made up of separate sections all stored individually on disc. These are then combined by the compiler to produce the finished program.



```
GSTC compiler version 1.02
Copyright © 1985 GST Holdings Limited
Command line from control file:
M:\PROG1 -P -M
main()
```

Compiling the source code to produce an assembly language listing

Step by step

THERE are several stages involved in writing an executable C program. Fortunately the whole process can be automated to a large extent, though this may require some effort setting up initially.

The separate stages are:

- Write the program using a text editor and save the source code in Ascii form.
- Compile the program – a utility called a preprocessor first expands the source code so it is readable by the compiler which then translates the listing into assembly language.
- Convert the assembly language program into a relocatable object code module using an assembler.
- Link the oject code with the necessary code modules from the C runtime library and translate it all into an executable machine code program.

This can take up to 10 minutes or more for a large program running from a floppy disc. If, after testing the program, it is found to have a bug the whole process will have to be repeated.

Which C Compiler?

A WIDE range of C compilers is available for the ST, and they have an equally vast price range starting at rock bottom with a free public domain version and rising to full development systems costing several hundred pounds. So what is the difference between them, and which is best?

The major differences are usually to be found in the number of utilities and library functions provided – useful routines written by the software company for you ready to slot into your programs. There are also minor differences in the compiler itself, but these

How to program in C

are probably less pronounced.

GST C is a budget C compiler costing as little as £15 at some retail outlets, and was used to write some of the programs in this section. It is supplied on a single disc along with a slim manual. Its major limitation is that it can't handle floating point arithmetic. However, it is surprising how rarely this facility is needed.

Mark Williams C – used for the rest of the listings – lies at the other end of the price range and is supplied on four discs packed full of ready made library functions and utilities. A 700 page manual is included too. It is expensive however, costing almost £150.

The lack of library functions and utilities in GST C isn't too serious a problem as you can always write your own, though it will require some time and effort which could otherwise be used to get on with developing your program. The lack of floating point maths is more serious: Imagine a spreadsheet used to run a large business that ignored the pence and just worked in pounds!

Making a start

WE will start off C programming with a very short, simple listing which just prints a message on the screen:

```
/* Program I */

main()
}
 printf("This is a C program\n");
 printf("The printf function ");
 printf("prints on the screen\n");
}
```

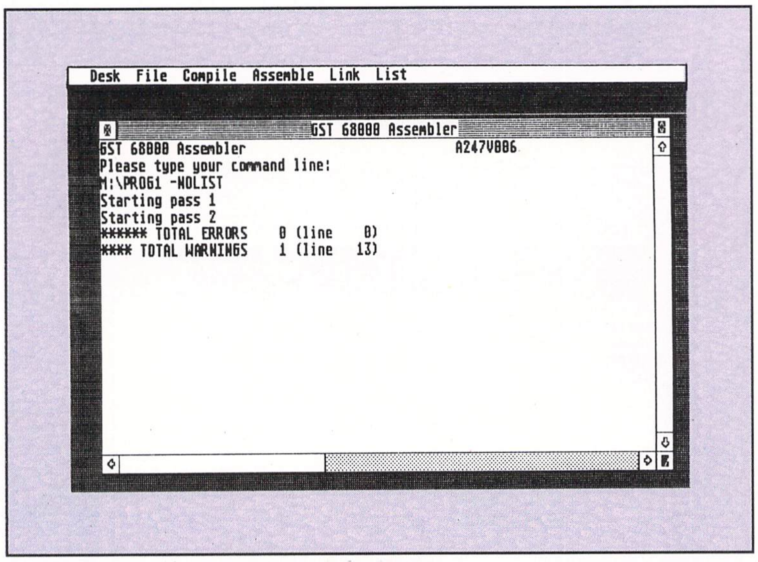
The first line of the program is a comment containing the program title. The /* tells the C compiler to ignore all the text following until it meets a */. It can then carry on compiling. Comments can be placed almost anywhere in the listing.

Every C program is composed of functions – they are easy to spot as they are followed by a pair of parentheses. There must be one called *main()* somewhere in the listing as this is where the program starts executing when it is run.

The chunk of code making up the *main()* function is contained within braces, and in our listing is made up of three *printf* functions. These are used to display the message on the screen.

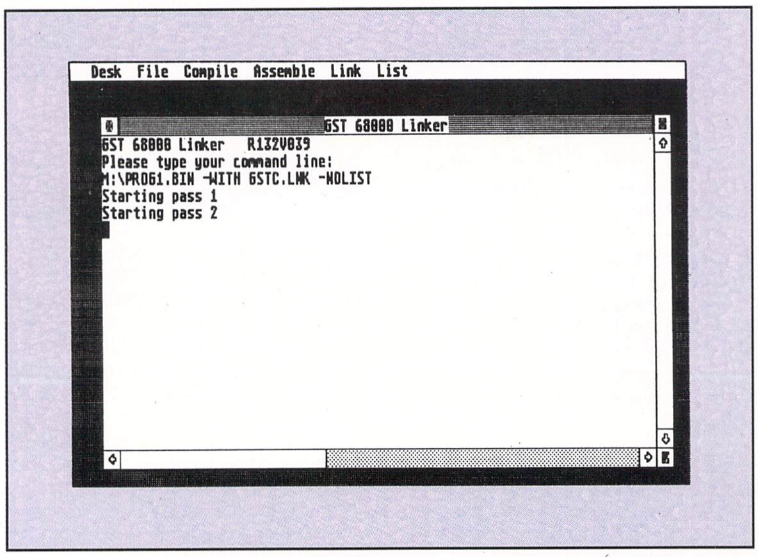
The text of the message is contained within quotes and each *printf* is terminated with a semicolon – this symbol separates all the statements and functions in C program listings.

Notice the \n at the end of the first and last text strings. This moves the cursor to the start of a new line on screen, otherwise the program would carry on printing the text on the same line each time as it does with the second and third strings.



```
GST 68000 Assembler                           A247V006
Please type your command line:
M:\PROG1 -NOLIST
Starting pass 1
Starting pass 2
****** TOTAL ERRORS     0 (line    0)
**** TOTAL WARNINGS     1 (line   13)
```

Assembling the assembly language listing to produce a relocatable object code module



```
GST 68000 Linker   R132V039
Please type your command line:
M:\PROG1.BIN -WITH GSTC.LNK -NOLIST
Starting pass 1
Starting pass 2
```

Linking the object code with the relevant code modules from the C runtime libraries

Glitches

IT'S quite rare that programs work perfectly first time, and there's often something we misstyped or forgot to include. These minor glitches will be highlighted by the compiler during the compilation process and an error message will be reported.

The messages will usually tell you what the error is and at what line it occurred. Take them with a healthy dose of salt though – they can be very misleading. It may be helpful to introduce a deliberate error into a C listing, in Program I say, and see what your compiler makes of it. The results can be surprising.

Assuming you've entered Program I correctly and compiled it, you may still have problems running it. What is likely to happen is that the program runs, ends, and returns to the desktop so quickly that you think it hasn't worked.

The solution is to pause after the program has finished so you can see the output on the screen. This is achieved by adding the line:

```
getchar();
```

to the end of the listing. This function gets a character from the keyboard – it waits for you to hit Return.

Some C compilers may also require an additional line telling them that the function *getchar()* is defined in the library STDIO.H. The command:

```
#include <stdio.h>
```

tells the compiler to include all the source code in the STDIO.H file when it sets about its work.

The complete listing now becomes:

```
/* Program I */

#include <stdio.h>

main()
}
 printf("This is a C program\n");
 printf("The printf function ");
 printf("prints on the screen\n");
 getchar();
}
```

The structure of a C program

SO far we've looked at the structure of a C program – it's made up of functions – and seen how to print text on the screen using *printf()*. The following short C listing should hold no surprises, and its task is simply to print a short message on the screen then wait for you to press the Return key:

```
/* Return key test utility */

#include <stdio.h>

main()
{
  printf("Hit the Return key...");
  getchar();
}
```

The first line is a comment – a note to remind us what the program does – and the next line tells the C compiler to include some special code stored on disc in a library called *stdio.h*. The main body of the program is to be found in the *main* function and is enclosed by braces – an open one marks the start and a close one marks the end.

The *printf()* function displays the quoted string on the screen, while the *getchar()* waits for you to press the Return key. These two functions are extremely powerful and we have only tapped a small proportion of this power so far. We'll see shortly that *printf()* can be used in many other ways by simply varying the parameters passed to it in brackets.

The variable nature of C

WE will move on now to slightly more complex aspects of the C language and discover how it handles numbers, variables and constants.

Although C is a fairly flexible language in many respects, it quite often imposes strict rules which must be obeyed. One of these is to insist that all variables, and their type, are declared before use, normally at the start of a function.

There are several different types of variable, but the most commonly used ones are *char, int* and *float*. The first denotes a single Ascii character, the second indicates an integer and the last is a floating point number. Other variables are *double, long, short, unsigned char, unsigned short, unsigned int, unsigned long* and pointer.

We'll restrict ourselves to integer variables for the moment and see how to master these. They are defined in a C program with lines like:

```
int length;
int width;
int area;
```

and once this is done we can assign a value to them and use them in calculations like:

```
length = 8;
width  = 5;
area   = length * width;
```

As you can see, this short chunk of code calculates the area of a rectangle eight units long by five wide. Having done this, how do we display the result? In fact, we use *printf()* yet again, but this time including extra parameters and formatting information within the brackets like:

```
printf("Area = %d ",area);
```

This time we are passing two parameters to *printf*, a string enclosed by quotes followed by a variable – *area*. Any per cent signs found within the string serve as markers showing the compiler where to place any extra parameters following it. In the line above the *%d* indicates a that a decimal number is to be placed at this point, and it's the one following the string – *area*.

We are now in a position to write a full C program incorporating simple integer variables:

```
/* Introducing variables */

#include <stdio.h>

main()
{
  int length, width, area;

  length = 8;
  width  = 5;
  area   = length * width;

  printf("Area = %d ",area);

  getchar();
}
```

Variable type	Storage size (bits)
char	8
double	64
float (or long float)	32
int (or short int)	16
long (or long int)	32
* (pointer)	32
short	16
unsigned char	8
unsigned short	16
unsigned int	16
unsigned long	32

The C variable types and storage size

The screen picture overleaf shows how to convert temperature measured in degrees Fahrenheit to degrees Celsius. It uses all the techniques we have discussed so far, but introduces a slightly more complex calculation into the program. This should't cause you any problems as you are quite likely to have come across similar statements many times before in Basic.

Repeating for a while

THIS Fahrenheit to Celsius conversion utility isn't too useful as it stands because it simply converts one fixed temperature to Celsius – 98F – and it would be much better if it converted all temperatures from freezing to boiling point. We can do this using a *while* loop like:

```
/* Fahrenheit to Celsius converter */

#include <stdio.h>

main()
{
  int f, c;

  f = 32;
  while ( f <= 212 ) {
    c = ( f - 32 ) * 5 / 9;
    printf(" %d F = %d C \n", f, c);
    f = f + 10;
  }
  getchar();
}
```

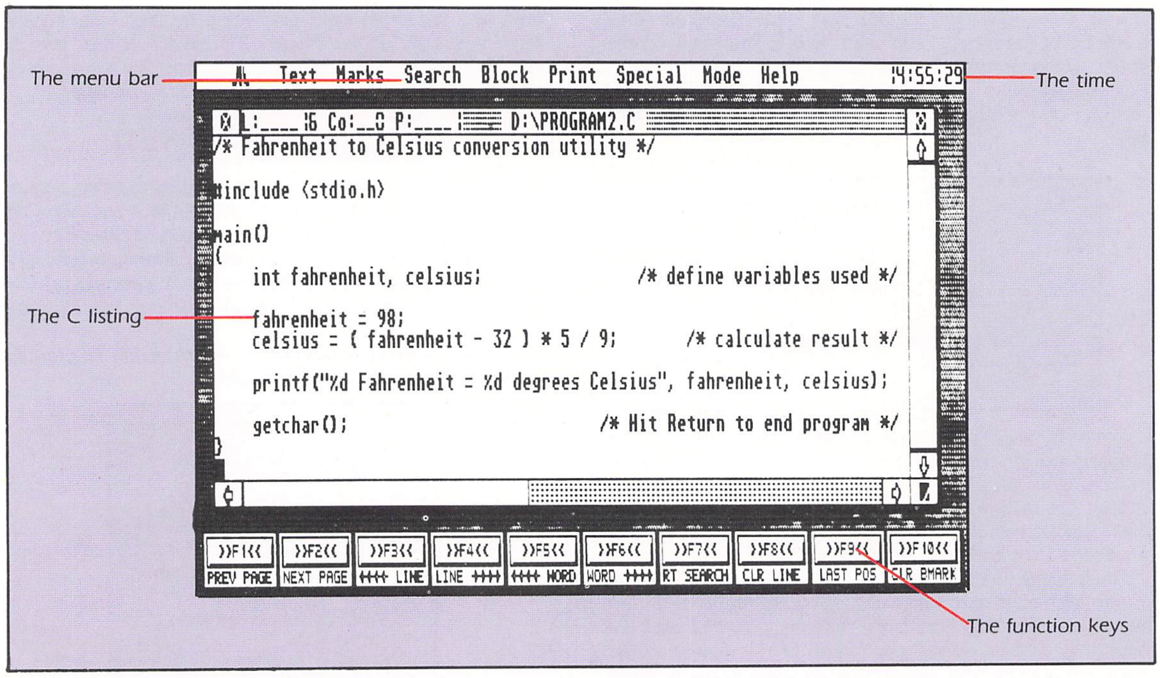
The *while* statement causes a section of program to be repeated while something is true. The section to be repeated – consisting of three lines in our program – is enclosed within braces and the statements within are indented slightly to aid readability.

There can be any number of statements within the braces and all will be repeated while the condition – the number, variable or expression within the brackets following the *while* – is true.

We repeatedly convert the temperature in degrees Fahrenheit to Celsius while *f* is less than or equal to 212 – the boiling point of water.

Notice that in the *printf()* string there are two occurrences

The structure of a C program



```
/* Fahrenheit to Celsius conversion utility */

#include <stdio.h>

main()
{
    int fahrenheit, celsius;                    /* define variables used */

    fahrenheit = 98;
    celsius = ( fahrenheit - 32 ) * 5 / 9;       /* calculate result */

    printf("%d Fahrenheit = %d degrees Celsius", fahrenheit, celsius);

    getchar();                                /* Hit Return to end program */
}
```

Using Tempus to enter a C listing

of *%d*. Each one indicates where the corresponding decimal integer variable following the string is to be placed.

We can significantly improve our Fahrenheit to Celsius conversion utility with a number of changes. The major one is to replace the *while* statement by a *for*. You may be familiar with the Basic version, and if you are the C version is likely to cause you a few headaches at first as it is so different. It's probably best to forget any pre-conceived ideas of how it works and treat is as an entirely different statement.

Written in very general terms the *for* statement looks like:

```
for ( expr1; expr2; expr3 )
  statement;
```

Taking the easy part first, *statement* can be a single C statement terminated by a semicolon, or a block of statements enclosed within braces like the one used with *while* in the temperature conversion utility.

The three expressions – *expr1, expr2,* and *expr3* – can be almost anything, so to illustrate the use of this new statement here is version two of our utility:

```
/* Fahrenheit to Celsius converter */

#include <stdio.h>

main()
{ int f, c;

  for ( f=32; f<=212; f=f+10 ) {
  c = ( f - 32 ) * 5 / 9;
    printf(" %d F = %d C \n", f, c);
  }
  getchar();
}
```

Notice how similar to the previous listing it is, but it is more compact. You can in fact usually convert a *while* to a *for* quite easily.

As before, the statement – or list of statements in braces – following the *for* is repeated. The first expression in the brackets following the *for* sets up the start condition, the second expression is the end condition and the final expression is executed each time the statements are repeated.

There is another small improvement that we can make, and you'll find this a common technique in C. The two following statements are identical:

```
f = f + 10;
f += 10;
```

When we want to add a value to a variable we can use this shorthand form to save typing. It also works with subtraction, multiplication and division too:

```
a = a * 5;
a *= 5;

b = b / 25;
b /= 25;

c = c - 2;
c -= 2;
```

Print formatting

WHILE we are improving our listing we may as well tidy up the output from this last program. If you compile and run it you should see something like:

```
32 F = 0 C
42 F = 5 C
52 F = 11 C
62 F = 16 C
72 F = 22 C
82 F = 27 C
92 F = 33 C
102 F = 38 C
112 F = 44 C
.
.
.
```

As you can see the output is slightly skew whiff and untidy. We can improve the appearance by telling the *printf()* function to pad out the numbers with spaces. We do this by inserting a number – the field width – in the *%d* like:

```
printf(" %3d F = %3d C \n", f, c);
```

The 3 following the per cent signs forces the numbers to be printed in a space three characters wide. If the number is less than three digits long it will be padded out with spaces. The resulting output is:

```
 32 F =   0 C      92 F =  33 C
 42 F =   5 C     102 F =  38 C
 52 F =  11 C     112 F =  44 C
 62 F =  16 C      .
 72 F =  22 C      .
 82 F =  27 C      .
```

which, I think you'll agree, is much more pleasing to the eye.

Function libraries in C

WE have seen that C is a language that is made up of functions like *main(), printf(), getchar()* and so on. Unlike some languages, it is also possible to add more functions of your own to those that C provides. In fact, without these extra add-on functions – either provided ready-written by the software company supplying the C compiler, or written by yourself – the language would be pretty useless.

Most computer languages have a rich supply of useful commands, statements, functions and procedures, but in contrast C has very few, and the missing bits are defined as functions which are provided in libraries. The most common of these is the *stdio.h* library that we include into the compilation process at the start of every program.

Usually many more are available, and particularly common are those for accessing Gem, which by the way, was written in C itself, though there is some machine code too. You'll find libraries like *osbind.h, aesbind.h, vdibind.h, gemdefs.h* and *linea.h* included at the start of many Gem-based programs. The manual supplied with the software will tell you what functions they contain.

Defining your own functions

WHAT we'll do now is to see how to define our own functions and look at their structure and properties. We'll start off with a very simple example, and later move on to something a little more complex.

The following C listing simply prints *Hello* on the screen, but it does so by calling a function which is defined at the end of the listing:

```
/* Defining functions 1 */

#include <stdio.h>

main()
{
  sayhello();
  getchar();
}

sayhello()
{
  printf("Hello");
}
```

The main part of the program is defined in the *main()* function and is enclosed by braces as usual. The first thing it does when the program is compiled and run is to execute the function *sayhello()* which can be found at the end of the listing.

This definition contains just one function – *printf()* which it executes, and as there are no more it returns to the *main()* function and continues program execution there.

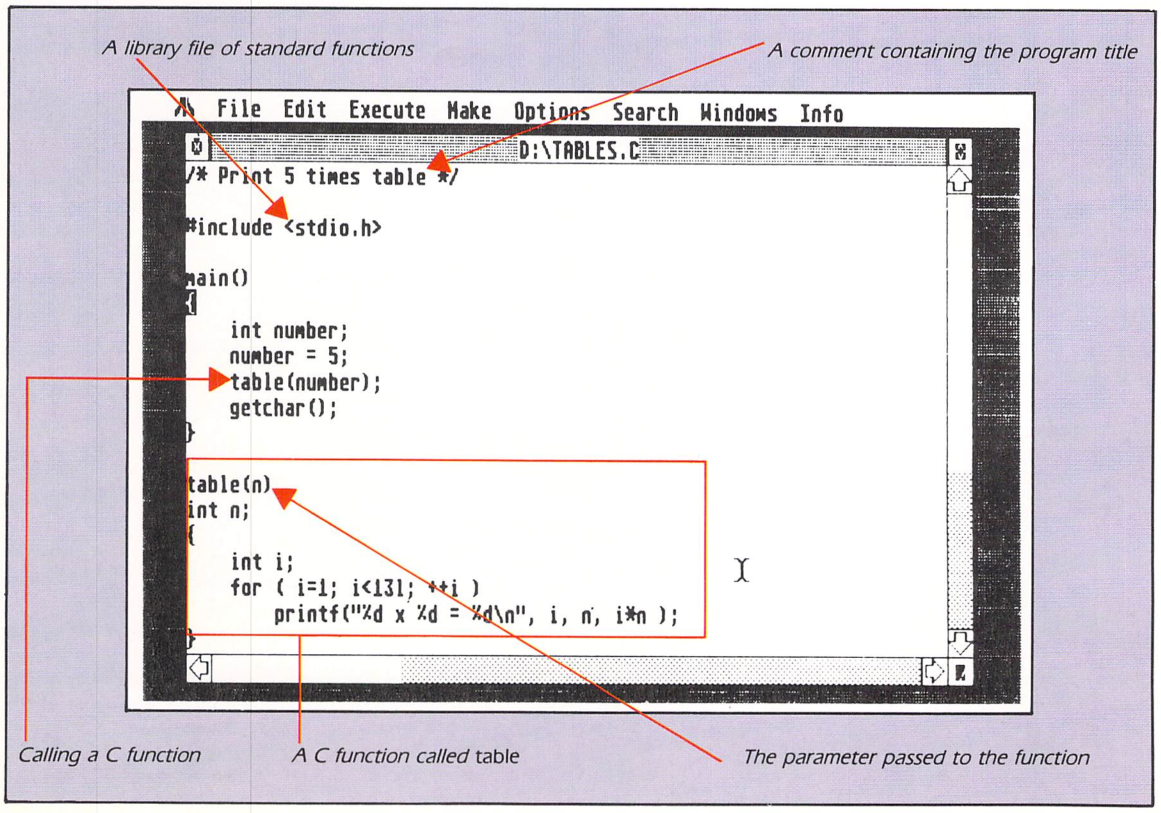
The second, and last, function in *main()* – *getchar()* – is executed, which as you already know, waits for you to hit the return key before continuing with the rest of the program. In fact, there aren't any more functions to execute, so the program ends.

Passing parameters

THIS is a fairly trivial example which doesn't do anything useful, so let's try something a bit more ambitious. The next program prints the five times table on the screen:

```
/* Print 5 times table */

#include <stdio.h>

main()
{
  int number;
  number = 5;
  table(number);
  getchar();
}

table(n)
int n;
{
  int i;
  for (i=1; i<13; ++i)
    printf("%d x %d = %d\n",i,n,i*n);
}
```

The first line of *main()* defines an integer variable called *number* and the following line sets its value to five – change this if you want to display other tables. The third line calls our function *table()*, passing it the parameter *number*, and


A library file of standard functions
A comment containing the program title
File Edit Execute Make Options Search Windows Info
D:\TABLES.C
Calling a C function
A C function called table
The parameter passed to the function

```
/* Print 5 times table */

#include <stdio.h>

main()
{
     int number;
     number = 5;
     table(number);
     getchar();
}

table(n)
int n;
{
     int i;
     for ( i=1; i<13; ++i )
          printf("%d x %d = %d\n", i, n, i*n );
}
```

Function libraries in C

the last line simply waits for a keypress again.

The function *table()* takes one parameter, and its type must be declared immediately after the function name. The body of the definition is contained between the braces and consists of a *for* loop to print out the five times table. The *printf()* function we've seen before, but never with this many parameters.

Each *%d* in the string to be printed is replaced by the corresponding parameter following it – the first by *i*, the second by *n* and, finally, the third by the result of the calculation *i* x *n*.

Returning a result

SO far we have seen how to define a function at the end of a listing, and how to call it from the main body of the program. Parameter passing has been covered, so we'll now move on to see how a result can be returned from a function.

To demonstrate this we'll return to familiar ground with the Fahrenheit to Celsius conversion program, but this time a separate function will be defined to calculate the new temperature:

```
/* Fahrenheit to Celsius conversion */

#include <stdio.h>

main()
{
  int f;
  f = 212;
  c = convert(f);
  printf("%d F = %d C\n", f, c);
  getchar();
}

convert(n)
int n;
{
  int temp;
  temp = ( n - 32 ) * 5 / 9;
  return(temp);
}
```

The function *convert()* is defined at the end of the listing as usual, and as before a parameter is passed which is defined as being integer. The body of the function consists of a variable definition followed by the calculation of the Celsius temperature. The final line is an instruction for the function to return the value *temp*, and this is used to set the value of *c* in *main()*.

It is important to note that only one value can be returned from a C function, and that the function exits when a *return()* statement is found. There can also be more than one *return()* too, as we'll see in our next program:

```
/* Returning values from functions */

#include <stdio.h>

main()
{
  printf("%d\n",test(1));
  printf("%d\n",test(2));
}

test()
int n;
{
  if ( n == 1 )
    return(5Ø);
  if ( n == 2 )
    return(1ØØ);
}
```

This short program prints two numbers – 50 and 100 – by calling the function *test()* first with the value one, then with two. An *if* statement tests the value of the parameter *n* and returns the appropriate value.

As you can see from the listing, the structure of the *if* statement isn't at all like Basic so take care.

The syntax is:

```
if (expression) statement

if (expression)
  statement1
else
  statement2
```

The expression can be almost anything which returns either true (non-zero) or false (zero), and the one we used was the test for equality using the double equals sign. Note that Basic uses a single equals sign with IF ... THEN and if you use this in C your programs won't work.

As there are only two possible values of the parameter *n, the function message()* could have alternatively been written like:

```
test()
int n;
{
  if ( n == 1 )
    return(5Ø);
  else
    return(1ØØ);
}
```

Local variables

WE have yet to discuss one more important aspect of functions, and that is variables. It must be remembered that all variables within a function are local – that is, even though the same variable name may appear in more than one function, the C compiler regards them as being completely different.

So if you had a variable called *number* which had the value five in *main()*, it could have a completely different value within another function.

For instance:

```
/* Local variables */

#include <stdio.h>

main()
{
  int number;
  number = 5;
  printf("number = %d\n",number);
  test(number);
  printf("number = %d\n",number);
  getchar();
}

test(number)
int number;
{
  number=number+1;
  printf("number = %d\n",number);
}
```

Run this to convince yourself that the *number* in *main()* is completely different to that in the function *test()*.

Its value is correctly printed as five both before and after the function call, yet inside the function the value of the (different but identically named) variable *number* is six.

To reinforce what we have learnt about functions, try to write one that prints the correct day when passed a number in the range 1 to 7.

Reading the keyboard

I/O functions

IT is about time we looked in more detail at how to get information into and out of C programs – input and output functions, or I/O for short. We have used a couple of functions already – *printf()* and *getchar()*.

We implemented the latter function as a pause by tagging it on to the end of our listings. It reads a single character from the keyboard, but works in a peculiar way. When you call *getchar()* you can type away until you hit Return, then, and only then, the first character is read.

The rest of them, up to and including the carriage return character itself, are stored in a buffer somewhere in memory and are read by subsequent calls to *getchar()*.

To read a character you would use a line like:

```
int c;
c = getchar();
```

where the variable *c* is an integer. Don't be tempted to make it a character variable because you're reading a character, as the value might not always lie in the 0 to 255 *char* range.

So far we have used the function *printf()* to display characters, strings of text and numbers on the screen. This is in fact a very complex function to use, and a simpler, more primitive one is also available called *putchar()*. It takes just one parameter, an integer, which it outputs as an Ascii code.

We can use *putchar()* to output single codes and characters that *printf()* isn't suitable for. Suppose we want to clear the screen by sending the codes Escape E to the VT52 emulator, we could use two *putchar()* function calls like:

```
putchar(27);
putchar('E');
```

Let's see these input and output functions in use. The listing below clears the screen using two *putchar()* functions calls, prints a prompt on the screen and then allows you to enter your name by successively reading characters from the keyboard by calling *getchar()*:

```
/* Get input from keyboard */

#include <stdio.h>

main()
{
  int c;

  /* Clear the screen */
  putchar(27);
  putchar('E');

  /* Print the prompt */
  printf("Enter your name:");

  /* Read the keyboard */
  c = 0;
  while ( c != '\n' )
    c = getchar();
}
```

Storing input

AS it stands, this demonstration isn't a great deal of use, as it won't store the input – each character read is simply discarded straightaway. We need to store the input so it can be recalled and manipulated later on.

In order to do this, C enables us to reserve space in memory by creating a character array. In some ways the process is similar to dimensioning an array in Basic. First we have to decide how much space we're going to need – in other words, how many characters is the user of the program likely to input? To be on the safe side we could allow for a 50 character name with:

```
char string[50];
```

The keyboard input routine used in the last listing could be expanded to store the characters in the array *string*:

```
c = 0;
i = 0;
while ( c != '\n' ) {
  c = getchar();
  string[i] = c;
  i = i + 1;
}
```

However, this is isn't good C programming style, and several shortcuts are normally taken. Everything in C has a value, and that value is the result of the last operation performed. So, the value of:

```
c = getchar();
```

is the value of the character read from the keyboard. This means we can compact the *while* routine to give:

```
while ( (c=getchar()) != '\n' )
```

Another common shortcut is to pre-increment and post-decrement variables. For instance, the following two lines mean exactly the same thing:

```
i = i + 1;
i++;
```

Combining all these shortcuts, and incorporating the whole lot in our first listing we end up with:

```
/* Get input from keyboard V2 */

#include <stdio.h>

main()
{
  char string[50];
  int c, i;
  putchar(27);
  putchar('E');
  printf("Enter your name:");
  i = 0;
  while ( (c=getchar()) != '\n' ) {
    string[i++] = c;
  }
  string[i] = '\0';
  printf("\nYour name is:%s",string);
  getchar();
}
```

Input functions

IT is also common C practice to split up programs into small sections or functions, each performing a simple, but specific task. We can do this with our example program by taking out the input routine and making it into a general function that can be included in any program where we wish to read the keyboard.

The main body of the code remains the same, with a *while* loop structure reading the keyboard and storing the characters in a *char* array. However, as we saw earlier, functions have their own private (local) variables and normally can't access those defined elsewhere in the program.

This means that we have to pass the character array *string* as a parameter. In fact, this is an advantage as it makes the function quite general and it will work with any program:

```
input(s)
char s[];
{
  int c, i;
  i = 0;
  while ( (c=getchar()) != '\n' )
    s[i++] = c;
    s[i] = '\0';
  }
```

Reading the keyboard

It would be called with a chunk of code like:

```
char string[50];
input(string);
```

and the text input would be stored in the *string* array. An alternative way of writing this would be to use a *for* structure, and some programmers prefer this as it is shorter, albeit only one line:

```
input(s)
char s[];
{
  int c, i;
  for (i=0; (c=getchar())!='\n'; ++i)
    s[i] = c;
  s[i] = '\0';
}
```

This can be compacted even further, and it is quite tempting to do so, but C programs are difficult enough to read and debug as it is without adding to the problem by squeezing as much code into as small a space as possible.

Limiting input

THE one problem with the routines we have looked at so far is that they don't check that the string will fit into the space allocated. We have allowed for a string of 50 characters, but suppose a novice accidentally holds a key down and then hits Return afterwards, inputting 60 or 70 characters. What will happen?

In this case the program is quite likely to crash, and possibly the ST will bomb out. Only pressing the reset button or switching off will enable you to regain control of your micro. C gives you a great deal of power and flexibility, but this means you have to be especially careful that your programs behave themselves and are bug-free.

What we have to do is somehow restrict the number of characters input so the limit of the character array isn't exceeded. This isn't a difficult task and involves just one extra check:

```
input(s,m)
char s[];
int m;
{
  int c, i;
  i=0;
  while ((c=getchar())!='\n' && i<m-1)
    s[i++] = c;
  s[i] = '\0';
}
```

Here an extra parameter *m* is passed to the *input()* function, and this tells it the maximum number of characters that it can place into the array *s*. In fact, we only store *m*-1 characters, as space needs to be left for the zero character terminating the string. If '\0' is omitted from the end of the string your program is quite likely to crash, or at the very least, display garbage.

Strings

STRING variables are handled by C in a completely different way to other languages like Basic, and none of the standard string manipulation functions is present in the language – you have to write your own. However, it isn't too difficult, and the code is usually quite short and compact.

A string is stored in memory as a string of characters terminated by a zero, or to be more precise, '\0'. You can initialise a string constant with the following declaration:

```
static char msg[] = "Hello there..."
```

The compiler will automatically place a zero at the end of the string, so its length is actually one more than the number of characters. You can also assign a string to a pointer, as we'll see when we discuss this complex topic.

An alternative method of incorporating a string into your program is to allocate space by defining a character array, and then to assign the string to that array like:

```
char s[50];
int i;
for (i=0; (s[i]=getchar)!='\n'; ++i);
s[i] = '\0';
```

Note that a zero is appended to the end of the input string. Also note that "A" and 'A' are two completely different things in C. The former is a one character string terminated by a zero, and the latter represents the Ascii code for the letter A – an integer.

String functions

STRING variables – or to be more precise, character arrays – are extremely useful structures in C programming. So as you might expect, functions are available for manipulating them. However, they aren't built in to the language, but are supplied as pre-defined library routines ready to be incorporated into your programs.

We could easily make use of these, but as many are quite simple and involve very little code it is good practice to have a go at writing them yourself. For instance, how long is a string? They can in fact, be any length, but frequently programs need to know exactly how many characters they contain. We'll write a short function to do this.

As we've seen, a string is always terminated by a zero character – or \0, which the compiler interprets as zero. So to find the length we simply start at the beginning and count the characters until we come to this zero end-of-string marker. It can be implemented like:

```
how_long(s)
char s[];
{
  int i;
  i = 0;
  while ( s[i]!='\0' )
    ++i;
  return(i);
}
```

The character array containing the string is passed as the parameter *s*. A counter, *i*, is set to zero and a *while* loop structure increments the counter while the array item is not equal to zero.

Another function we may require is one that reverses a string. We can write a function *backwards()* that reverses any character array passed to it. The function will need to know the length, and this can either be passed as a parameter or found using *how–long()*:

```
backward(s,len)
char s[];
int len;
{
  int i,j;
  char temp;
  i = 0;
  j = len-1;
  while ( i<j ) {
    temp = s[i];
    s[i++] = s[j];
    s[j--] = temp;
  }
}
```

Case conversion

WE will continue to explore character arrays – often used for storing strings – and write some useful functions to manipulate them in various ways. Although many of the routines we'll look at are available as ready made library functions, it is usually a trivial task, and good practice too, to write your own versions.

One of the things we often wish to do with a chunk of text stored in a character array is to check that it contains either all upper case letters or just lower case ones, and convert those that aren't to the correct case. A function to do this would have to examine each character and test it to see if it was in the range A to Z for upper case and a to z for lower case.

The following program demonstrates how to input a line of text from the keyboard and convert any upper case letters to lower case. It is written using GST C, but contains nothing unusual, so should be able to be compiled by most C packages:

```
/* Convert input to upper case */
/*       Written in GST C       */

#include <stdio.h>
#define MAXLEN 100

main()
{
  char string[MAXLEN];
  input(string,MAXLEN);
  lower(string);
  printf("\n%s",string);
  getchar();
}

input(s,m)
char s[];
int m;
{
int c,i;
for (i=0;(c=getchar())!='\n' && i<m-1;)
  s[i++] = c;
s[i] = '\0';
}

lower(s)
char s[];
{
  int i;

  for (i=0; s[i]!='\0'; ++i)
    if (s[i]>='A' && s[i]<='Z')
      s[i] += 32;
}
```

Enter it, compile it and run it to see it in action. Try typing a mixture of upper and lower case letters and see that it converts them all to lower. You've seen input routines before, so *input(string)* should hold no surprises. It reads the input and places the characters in the character array *string* which is passed as the parameter *s*.

The function we are interested in is the one at the end called *lower()* as it is this that does the actual conversion. A *for* loop structure is used to scan through the array character by character. An *if* statement tests the current character to see if it is upper case, and if it is, 32 is added to the Ascii code – the increment needed to make an upper case letter lower case.

The line:

```
for (i=0; s[i]!='\0'; ++i)
```

sets up the loop, and initialises the loop counter *i* to zero. The end of the string is tested for by the middle conditional statement, which checks that the character is not equal to the zero end-of-string marker. The final statement in the brackets increments the loop variable *i*.

The following *if* condition tests if the character is greater than or equal to A and less than or equal to Z (the double ampersand sign is C's equivalent to Basic's logical AND operator). In other words, it asks whether the character is a capital letter:

```
if (s[i]>='A' && s[i]<='Z')
```

The final line of this function, which is executed if the *if* condition evaluates to true, is an interesting feature of the C programming language, and at first sight it appears to be very strange indeed:

```
s[i] += 32;
```

What this does is to add 32 to the array element *s[i]* and the line is exactly equivalent to the alternative, and more traditional way of writing the expression:

```
s[i] = s[i] + 32;
```

The only difference is that the latter version takes slightly longer to type. The accompanying panel shows some similar C shortcuts and abbreviations along with their full expanded versions.

Abbreviation	Meaning
x = y	x = y
x += y	x = x + y
x –= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x >>= y	x = x >> y
x <<= y	x = x << y
x &= y	x = x & y
x ˆ= y	x = x ˆ y
x |= y	x = x | y

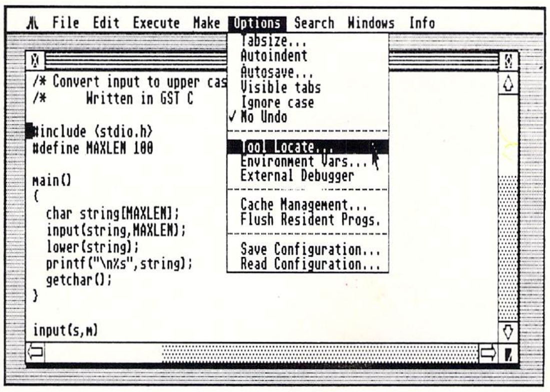
Common abreviations and their meaning

One final point to note with this listing is the *#define* at the start. This tells the compiler to replace every occurrence of *MAXLEN* with the number 100. If at any time we want to change the maximum length of input, we simply alter the 100 following *MAXLEN*.

The advantage of this is that we don't have to search through the program listing changing all the maximum lengths, which would be the case if we had used the actual number 100 in the program.

Numeric input

WE will move on to see how to input numbers into our C programs rather than character strings. In fact, the technique involved uses a large chunk of code from our last program.



```
/* Convert input to upper cas
/*      Written in GST C

#include <stdio.h>
#define MAXLEN 100

main()
{
  char string[MAXLEN];
  input(string,MAXLEN);
  lower(string);
  printf("\n%s",string);
  getchar();
}

input(s,m)
```

Laser C, although very expensive, can reduce compile-link-run times from minutes to seconds

Case conversion

What we have to do is to input the number into a character array as a string of Ascii characters. We can then scan the array and calculate the number from the Ascii codes of the digits stored there.

The program here allows you to input a decimal number, which it then prints out in hexadecimal using the *%x* conversion specification in the function *printf()*:

```
/* Convert decimal to hexadecimal */
/*        Written in GST C         */

#include <stdio.h>
#define MAXLEN 10

main()
{
  char string[MAXLEN];
  int number;
  input(string,MAXLEN);
  number = val(string);
  printf("\nHex = %x ",number);
  getchar();
}

input(s,m)
char s[];
int m;
{
int c,i;
for (i=0;(c=getchar())!='\n' && i<m-1;)
  s[i++] = c;

s[i] = '\0';
}

val(s)
char s[];
{
  int i,n;
  n = 0;
  for (i=0; s[i]!='\0'; ++i)
    n = 10*n+(s[i]-'0');
  return(n);
}
```

The input routine we've seen before so I'll say no more about this. The part we are interested in is the function called *val()* at the end of the listing, as it is this which scans the string of digits and converts it into a number.

The way it works is to initially set the number *n* to zero. Then a *for* loop scans the string from start to finish – remember the end of the string is marked by a zero character. The number *n* is multiplied by 10 and the next digit – calculated by subtracting Ascii zero from the Ascii code of the character – is added to the result. This process is repeated until there are no more characters. Finally *n* is returned as a result of the function with *return(n)*.

Operator	Function
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Remainder after division
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
!	Logical NOT
&&	Logical AND
||	Logical OR
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
>>	Shift bits right
<<	Shift bits left
~	Ones complement

C operators

Counting up

STILL staying with strings, but this time returning to pure text rather than numeric input, we'll look at a new function to count the number of words in a line of input – the basis of a simple word counter for word processors.

We can use the same input routine as in our previous examples, and once we've got the text stored in a character array we can scan it and count the number of words quite easily.

All words start with a letter, so what we do is start at the beginning of the array and examine each character. When we find either an upper or lower case letter then that is the start of the first word.

We now have to skip the rest of the characters in the word and look for its end, marked by either a space or the end of the input string – a zero marker. If a space is found, we look for the start of the next word, and so on until we come to the end of the input string.

Here is the whole listing:

```
/*   Word counter    */
/* Written in GST C */

#include <stdio.h>
#define MAXLEN 100

main()
{
  char string[MAXLEN];
  int words;
  printf("Enter some text: ");
  input(string,MAXLEN);
  words = count(string);
  printf("\nThere are %d words",words);
  getchar();

}

input(s,m)
char s[];
int m;
{
int c,i;
for (i=0;(c=getchar())!='\n' && i<m-1;)
  s[i++] = c;
s[i] = '\0';
}

count(s)
char s[];
{
  int i,words;
  words = i = 0;
  while ( s[i]!='\0' )
    if ((s[i]>='A' && s[i]<='Z') ||
 (s[i]>='a' && s[i]<='z'))
    {
      ++words;
      while (s[i]!=32 && s[i]!=0)
        ++i;
    }
    else
      ++i;
  return(words);
}
```

The important function is *count()* at the end of the listing. After declaring the variables, both *words* and *i* are set to zero. Then a *while* loop structure contains the main body of the code.

The *if* statement tests whether the character is greater than or equal to A and less than or equal to Z – in other words a capital letter – or greater than or equal to a and less than or equal to z – a lower case letter. If it is, *words* is incremented and a *while* loop skips the characters in the word, looking either for a space – Ascii 32 – or a zero – end of string marker.

If the *if* test fails and the character is not a letter, the pointer *i* is incremented so that *s[i]* points to the next character in the string.

Floating point variables

Printing numbers

WE have looked at several different variable types commonly used in C programming, but two we haven't really explored in any depth are floating point variables and numbers. High level languages like Basic handle them just like ordinary integers and you can usually manipulate floating point variables and numbers without changing your program significantly.

Many Basics like ST Basic distinguish integers from floats – sometimes called real numbers – by appending a per cent sign to the variable name. So *num%* would be an integer or whole number but *num* would be a float or real number. STOS Basic is unusual in that it assumes you are using integers and you must append a hash sign to indicate a real number.

Unlike Basic, in C programming we don't use a per cent or hash sign, but we do have to tell the compiler which variables in our program are integer and which are floating point. This is done by defining the variables at the start of *main()* or in whatever function they are used like:

```
float height;
double length, width;
```

The first example defines the variable *height* as being a floating point number. The second example defines *length* and *width* as being double precision floats – meaning you can have more digits after the decimal point, hence a greater degree of accuracy.

To demonstrate the use of this new type of variable here is a short program which prints out the numbers up to 10 along with their square roots:

```
/* Floating point output */
/*    Mark Williams C    */

#include <stdio.h>
#include <math.h>

main()
{
  float n,r;

  printf("Number.....Root\n");
  for ( n=1; n<11; n+=1 ) {
    r = sqrt(n);
    printf("%f = %f\n",n,r);
  }
}
```

This is written using Mark Williams C, an expensive but comprehensive C development system. In past example listings the budget-priced GST C was used, and while this is excellent for most tasks its one main drawback is that it can't handle floating point arithmetic. Decimal numbers are out and you are restricted to whole numbers – which, in practice, is rarely a handicap.

Some compilers, Mark Williams for instance, require that you specifically tell it to include its floating point routines into the final code. If you forget to do so when compiling the source code confusing error messages may be displayed, as it assumes you are using integer maths throughout.

The compiler command line for Mark Williams is:

```
cc -f program1.c -lm.
```

The –f tells the compiler to include the floating point *printf()* function instead of the standard one. The reason it is not normally included is that it requires extra code, thus making the program slightly bigger. The –lm at the end tells the linker to link in the maths library as we call a built-in function called *sqrt()* to calculate the square root.

In addition to this, the maths library must be included in the compilation process with an *include* command at the start of the listing.

A further point worth noting is that some C compilers insist that you include a decimal point when assigning a value to a floating point variable like:

```
n = 5.0 * 6.0
```

and not:

```
n = 5 * 6
```

You'll notice from the listing (and the output if you enter and run this program) that the variable *n* runs from 1 to 10 in steps of one, and only whole numbers are used. You might think that *n* could therefore be defined as being an integer, but if you try it all the square roots come out as being zero. What has happened is that the function *sqrt()* expects a float to be passed to it as a parameter and you provide it with an integer. It gets confused and assumes it's a float with a value of zero.

Finally, take a look at the *printf()* function. You'll see that it includes the %f conversion specification. This instructs the print routine to place a floating point number at this point in the string being output.

We can add a couple of parameters to this – placed between the per cent and the f – to tell it how many decimal places to print and how much space it has got. For instance,

Numeric output

SO far we have looked at ways of inputting numbers into our C programs, and this has involved entering a string and then converting the Ascii characters into a number. Although we have also seen how to output numbers using the *printf()* function, it is good practise to have a go at writing your own output routine.

One method involves the reverse process – converting the number into a string and then printing this character by character:

```
printd(n)
int n;
{
  char string[10];
  int i;
  i = 0;
  while ( n > 0 ) {
    string[i++] = (n % 10) + '0';
    n /= 10;
  }
  while ( i >= 0 )
    putchar( string[i--] );
}
```

This function can be called with something like *printd(12345)* and it will print the number on the screen. It works by successively dividing the number by 10 and storing the digits in a character array. When the number reaches zero the string is complete, but unfortunately the digits are in reverse order. A *while* loop is used to output the characters and *putchar()* directs them to the screen.

You might think that this is a fairly compact and efficient routine, but in fact, it can be made much simpler and shorter by implementing it as a recursive function:

```
printd(n)
int n;
{
  if ( n > 9 )
    printd( n/10 );
  putchar( (n % 10) +'0' );
}
```

This new version of *printd()* is only possible because of the way C implements local variables each time a function is called. If the number passed to *printd()* is greater than nine it repeatedly calls itself and divides the number by 10 until it gets a number small enough to print.

Floating point variables

%5.2f would mean print the floating point number in a space five characters wide – padding out with spaces if it's smaller – with just two digits after the decimal point.

For a much neater output alter the *printf()* function so that it reads:

```
printf(" %2.0f  =  %4.2f\n",n,r);
```

The maths library header MATH.H is a simple C listing that defines a few constants and informs the compiler that certain maths functions will be used. The only task it performs in our program is to tell the compiler that the function *sqrt* returns a double precision floating point number.

By default, all functions return integer results so a *double sqrt ()* at the start of the listing must be entered.

Inputting floats

WE have examined several input routines for entering strings and integer values, and now it is time we saw how to enter decimal numbers into floating point variables. What we'll do is to input the number as a simple string, and once it is safely stored in memory we'll convert it into a number. We've seen the string input routine before, so let's not dwell on that. Here is the function which converts that string into a real number:

```
double getfloat(s)
char s[];
{
  int i;
  double n,fact;
  n = 0.0;
  for ( i=0; s[i]!='.'; ++i )
    n = n*10.0 + (s[i]-'0');
  fact=0.1;
  ++i;
  while ( s[i]!='\0' ) {
    n = n + fact*(s[i++]-'0');
    fact = fact/10.0;
  }
  return(n);
}
```

The function is called *getfloat()* and the string is passed as a parameter. The function returns a floating point number so it must be defined as being either a float or – as in this case – a double (float). If you don't do this the compiler will assume the result returned by the function is an integer and you'll end up with a nonsense value for the input.

The function is in two parts. The first half works out the value of the whole number part of the input and the second half calculates and adds the decimal digits. Here is the routine incorporated into a simple program:

```
/*    Floating point input     */
/* Written in Mark Williams C */

#include <stdio.h>

main()
{
  char string[20];
  double number , getfloat();
  input(string);
  number = getfloat(string);
  printf("Number = %f\n",number);
}

input(s)
char s[];
{
  int i,c;
  for ( i=0; (c=getchar())!='\n'; )
    s[i++] = c;
  s[i] = '\0';
}

double getfloat(s)
char s[];
{
  int i;
  double n,fact;
  n = 0.0;
  for ( i=0; s[i]!='.'; ++i )
    n = n*10.0 + (s[i]-'0');
  fact=0.1;
  ++i;
  while ( s[i]!='\0' ) {
    n = n + fact*(s[i++]-'0');
    fact = fact/10.0;
  }
  return(n);
}
```

What this doesn't allow for is negative numbers, so you might like to modify the function to cope with this situation. All you need to do is examine the first character and set a flag if it is a minus sign. If one is found return *–n* instead of *n*.

Register variables

THERE is a special type of variable available to C programmers called a register variable. Normally variables are allocated space in memory to hold their name and value, the amount of space depending of the length of name and type of variable, such as integer, float or string.

However, register types are completely different in that they make use of the 68000 processor's internal registers to hold the variable's value. This imposes certain restrictions on its type, but the advantage is that they can be accessed at lightning speed.

The only data type available in register form is integer, as the 68000 chip's registers can't manipulate floating point numbers. Also only a limited number of registers is available so this too, places a limit on the number you can define in your program. It is best to reserve them for speed sensitive parts of your code and to restrict them to very heavily used integer variables.

To convert an ordinary integer to the register type simply precede its definition with *register*. To demonstrate their use the following short program takes 20.93 seconds to execute:

```
/* Speed test */

#include <stdio.h>

main()
{
  int i , j;
  for ( i=0; i<20; ++i )
    for ( j=0; j<30000; ++j ) ;
}
```

If, however, we insert the *register* command before the *int* like:

```
register int i , j;
```

execution time is reduced to 13.23 seconds – around two thirds of the original time and quite a significant saving.

Peeking and poking

New variables

ONE important topic we have yet to cover in this section on C programming is pointers. It can be a confusing subject for newcomers as it introduces ideas which may be quite foreign, but in fact, it is fairly straightforward once you get the hang of it.

Although there are significant differences, there are also several similarities to Basic's PEEK and POKE, as we'll see. Wherever possible we'll show the Basic equivalent to the C code.

Pointers are a different type of variable. They aren't at all like the character, integer or floating point variety we've seen so far. Pointers, as their name suggests, are variables that point to items or objects stored in the computer's memory.

They, in fact, hold their address. These objects may be other variables, strings or arrays. Suppose *x* is an integer variable and its value is five. This can be defined like:

```
int x;
x = 5;
```

If *ptrx* is a pointer (and without going into detail about how it was created) then using the ampersand operator we can set it to point to the address at which the value of *x* is stored with the line:

```
ptrx = &x;
```

The quivalent statement in Basic would be:

```
ptrx = VARPTR(x)
```

Now that *ptrx* is pointing to *x*'s address what can we do with it? Well, *x*'s value can be accessed in a similar manner to the way in which Basic's PEEK and POKE operate. The following line sets *y* equal to the value of *x* – in this case, five:

```
y = *ptrx;
```

The asterisk indicates that you are accessing the contents of the item pointed to by *ptrx* and it vaguely resembles Basic's:

```
y = PEEK(ptrx)
```

where *ptrx* is an address. The value of *x* can also be modified using a similar process:

```
*ptrx = 6;
```

Notice that the pointer is now on the left hand side of the assignment statement and this means that the contents of the item pointed to by it is equal to whatever is on the right hand side – in this case, the integer value 6. This is like the Basic statement:

```
POKE ptrx,6
```

As a further example, listing I demonstrates the use of pointers by using them to swap the values of two variables. Notice that the two pointers are defined in line two of *main* as being integers (meaning the objects they point to are integers).

At first sight the third statement from the end is a strange one. It sets the contents of the integer pointed to by *ptrx* to the value stored at *ptry*.

In Basic this would look like:

```
POKE ptrx,PEEK(ptry)
```

It must be stressed that the Basic equivalents are similar but by no means identical. C takes into account the object the pointer is pointing at and adjusts its operation accordingly. Integers, floating point variables and character arrays all take up different amounts of space in the memory. C will automatically alter the number of bytes read or written to in pointer operations.

```
/* Swap variables */
/*   Prospero C    */

#include <stdio.h>

main()
{
    int x , y , temp;
    int *ptrx , *ptry;
    x = 3;
    y = 4;
    printf("x = %d , y = %d\n",x,y);
    ptrx = &x;
    ptry = &y;
    temp = x;
    *ptrx = *ptry;
    *ptry = temp;
    printf("x = %d , y = %d\n",x,y);
}
```

Listing I

Character pointers

THIS isn't the limit of the use of pointers, and we have only just scratched the surface. Pointers can aim at any object, so they can just as easily point to a string, which is in fact, essentially an array of characters. So the following statements are quite valid:

```
char *string
string = "Some text..."
```

In this case the pointer *string* points to – or in other words, holds the address of – the start of the string. Now the pointer is defined as being a *char*. This type of assignment is identical to the variety we've seen before when defining string variables. The following two statements are the same in C:

```
char string[] = "Hello...";
char  *string = "Hello...";
```

In order to be able to perform simple arithmetic operations on pointers C must be told what types of object they are pointing to. For instance, if the pointer *ptr* points to the string "ABC" it will hold the address of the first character – the start of the string. If the pointer is incremented with ++*ptr* it will point to the second character. If it is incremented again it will then point to the third and so on.

The following program defines the pointer *ptr* and sets it to point to the string "ABC". The three characters of the string are altered one by one so that the string becomes "XYZ":

```
/* Character Pointers */
/*     Prospero C     */

#include <stdio.h>

main()
{
    char *ptr;
    ptr = "ABC";
    printf("%s\n",ptr);
    *ptr = 'X';
    ptr++;
    *ptr = 'Y';
    ptr++;
    *ptr = 'Z';
    ptr--;
    ptr--;
    printf("%s\n",ptr);
}
```

In this case incrementing the pointer makes it point to the

Peeking and poking

next character in the string, which is essentially an array of characters. If the pointer was an integer type pointing to an array of integers it would be incremented by whatever amount is necessary so that it points to the next integer.

String functions

THE following short C program demonstrates how to compare two strings to see if they are the same. The code does not use pointers:

```
/* Copy strings */
/*  Prospero C  */

#include <stdio.h>

main()
{
  char astring[] = "Hello...";
  char bstring[] = "Goodbye!";
  int test;
  test = compare(astring,bstring);
  if ( test )
    printf("\nEqual\n");
  else
    printf("Not equal");
}

compare(a,b)
char a[], b[];
{
  int i;
  i = 0;
  while ( a[i] == b[i] )
    if ( a[i++] == 0 )
      return(1);
  return(0);
}
```

The function *compare()* at the end of the listing is the one that does the comparing and it returns a value of 1 if the strings are the same, otherwise zero is returned.

The strings are passed as character arrays using code that we've seen several times before so there's nothing unusual here.

The function can be rewritten using pointers:

```
compare(a,b)
char *a, *b;
{
    while ( *a++ == *b++ )
        if ( *a==0 && *b==0 )
            return(1);
    return(0);
}
```

We have seen the post increment function before, but here it is worth noting that the increment in **a++* refers to incrementing the pointer and not the contents of the object pointed to. As you can see, this version of *compare* is slightly more compact and is just as readable.

In a similar fashion we can also copy strings using either character arrays or pointers. As before, the pointer version is more concise.

Here is the standard non-pointer version:

```
copy(a,b)
char a[],b[];
{
    int i;
    i = 0;
    while ( a[i] != 0 ) {
        b[i] = a[i];
        ++i;
    }
}
```

And the pointer version looks like this:

```
copy(a,b)
char *a,*b;
{
    while ( (*b = *a) != 0 ) {
        a++;
        b++;
    }
}
```

While this may seem like an improvement it can be shortened. The post-increment instruction can be added to the pointers in the *while* test:

```
copy(a,b)
char *a,*b;
{
    while ( (*b++ = *a++) ) ;
}
```

This shows just how compact C listings can become. There is a temptation to produce extremely compact, and consequently, almost totally unreadable code.

Stringing along

POINTERS can be employed for quite a wide variety of functions, so as a further example of their use we'll write a simple function to return the length of a string.

A string is a character array and the function written as normal without using pointers would look like:

```
getlen(s)
char s[];
{
    int i;
    i = 0;
    while ( s[i] != '\0' )
        i++;
    return(i);
}
```

The counter *i* is set to zero and each character is examined to see if it is the end of string marker – zero. When it is found the length is returned. You would call the function with a line like:

```
length = getlen(string);
```

There's nothing wrong with this function, but as with any programming task there is always an alternative way of performing the same operation. If we rewrite this making use of pointers our first attempt might look something like:

```
getlen(s)
char *s;
{
    char *p;
    p = s;
    while ( *p != 0 )
        p++;
    return(p-s);
}
```

As we have seen before, we can compact this code quite significantly. The pointer declaration can be combined with the following assignment line. The pointer can be incremented in the *while* test and we don't, in fact, need the test itself. Here is the compacted version:

```
getlen(s)
char *s;
{
    char *p = s;
    while ( *p++ ) ;
    return(p-s-1);
}
```

An introduction to Midi

Compiled by Ian Waugh

THIS section is for everyone who doesn't know what the Midi sockets are for, and for those who hanker after making music even if they don't consider themselves to be much of a musician.

New technology

MUSIC and indeed Midi, on the ST is a consuming, relaxing and rewarding pastime. However, it can be a very confusing area to get into, even for the accomplished traditional musician, with lots of new technology to take on board and many new terms to learn.

We will take you on a trip from the very beginnings to quite advanced composition and music techniques, such as sequencing, scorewriting and voice editing. If you are a newcomer both to music in general and Midi in particular some of the global terms applied within this field may be at first confusing. These will be explained where necessary, though it is not possible to do so every time such a term appears.

One of the prime aims of this section is to de-mystify Midi. Like sport, music is for everyone. It is often more fun to play something yourself, however badly, than it is to listen to a master musician. Through Midi it is possible for anyone – even if they only have a small amount of musical ability – to produce musical arrangements and have lots of fun in the bargain. Above all, music should be enjoyable and not a chore.

What is Midi?

LET us get Midi – an acronym for Musical Instrument Digital Interface – under our belt first. Assuming that you know what a musical instrument is, an interface is something which joins two things together. So a musical instrument interface is something which joins two musical instruments together.

Digital describes the way the interface works and in case you have just this minute unpacked your ST, it basically means numbers. That is, the signals used to convey information from one instrument to another – via the interface – are digital signals, 1s and 0s, exactly the same as the numbers floating around inside your ST.

The Midi specification, drawn up by the world's major musical instrument manufacturers in 1983, specifies the data format – the numbers – to be used to transmit various musical messages. For example, there is a Midi message meaning turn a note on – the musical equivalent of pressing a key on a keyboard – and a message meaning turn it off – take your finger off the key. Using just these messages we could program the computer to play a tune on a Midi-compatible instrument by making it transmit a series of Note On and Note Off messages through the ST's Midi Out socket.

There are more aspects of a musical performance to consider than the Note Ons and Offs. What about dynamics? On an acoustic piano, the harder you hit a key the louder the note sounds. Many synthesisers create this effect by using a velocity sensitive keyboard. This responds to the speed with which you press down a key and converts it into velocity (or loudness) information.

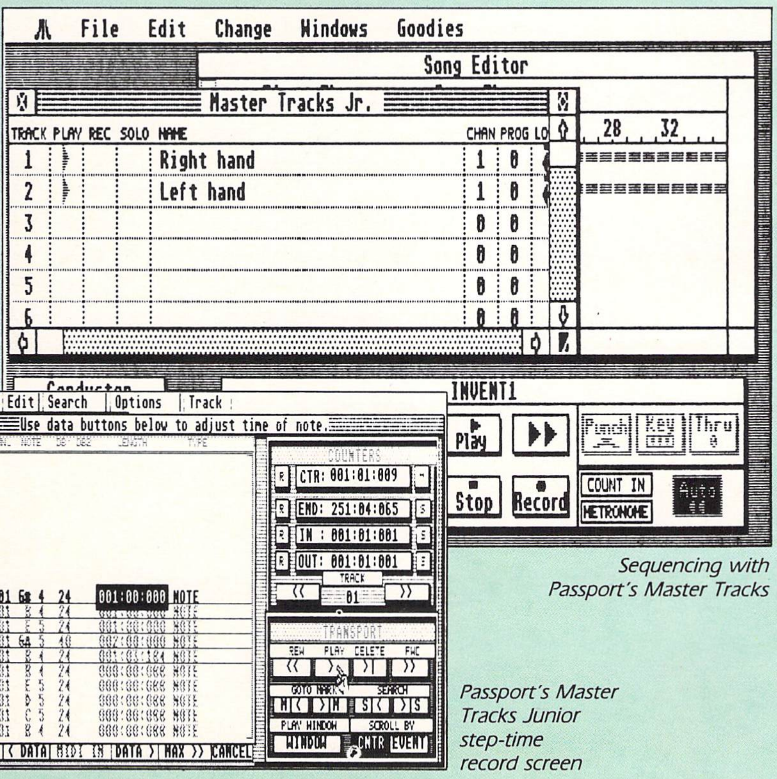
Synthesisers can produce lots of different sounds and you can change these over Midi by sending Program Change messages. You can control a synthesiser's pitch bend wheel and modulation wheel and activate any of a number of controllers to switch vibrato on and off and control a sustain footswitch, for example.

There are more Midi messages than these, but all consist of a series of numbers – that's basically what Midi is about.

Like a tape recorder

IF you connect an instrument to your ST through the Midi sockets and play some notes the relevant Note On, Note Off and Velocity messages zap along the cable into the computer. If you were running a sequencing program, this information would be stored as numbers. To reproduce what you have played you just transmit the messages from the computer back to the instrument.

You can see that in such a case the computer is acting very much like a tape recorder. In fact, much Midi software adopts a tape recorder approach, with controls labelled Record, Play, Fast Forward, Rewind and so on. Most musi-



```
1   Right hand   1  0
2   Left hand    1  0
3                0  0
4                0  0
5                0  0
6                0  0
```

```
Use data buttons below to adjust time of note.
BAR  BT  PLS CHNL  NOTE  DB DB2    LENGTH       TYPE
001:01:009 01 G# 4  24    001:00:000 NOTE
002:01:009 01 B  4  24    001:00:000 NOTE
003:01:009 01 E  5  24    001:00:000 NOTE
004:01:009 01 G# 5  40    002:00:000 NOTE
006:01:017 01 B  4  24    001:03:184 NOTE
008:01:017 01 B  4  24    000:00:088 NOTE
008:01:113 01 E  5  24    000:00:088 NOTE
008:02:017 01 D  5  24    000:00:088 NOTE
008:02:113 01 C  5  24    000:00:088 NOTE
008:03:017 01 B  4  24    000:00:088 NOTE
```

```
CTR: 001:01:009
END: 251:04:065
IN : 001:01:001
OUT: 001:01:001
01
```

Sequencing with Passport's Master Tracks

Passport's Master Tracks Junior step-time record screen

An introduction to Midi

cians are familiar with the principles of multi-track recording and transferring that familiarity to the software makes it easy to understand.

The important thing to realise, however, is that the computer is only storing digital signals – numbers – and not analogue or audio signals which tape recorders use. As a computer user, you'll appreciate how easy it is to manipulate numbers.

For example, to play a tune faster using a tape recorder you must increase the speed at which the tape runs past the playback head. This, unfortunately, also results in an increase in pitch. Tape recorders with varispeed controls work this way.

Using your ST, all you have to do to increase the speed is to transmit the numbers a little faster. You're not altering the numbers themselves, so the pitch remains the same. If you want to increase the pitch that's easy to do, too. Simply add a constant value to all numbers representing a pitch.

One more example. If you have a tape recorder you'll know some recordings are noisy with tape hiss. This is a result of recording signals at a very low level. Multi-track machines can record several different music parts on different tracks. To free a track so you can record another part on it, it's common practice to mix two or more tracks together by recording them on to a single track, a process known as bouncing. The more times you bounce, the more noise you add to the system. This is because you are working with analogue signals.

As soon as you switch to numbers, signal degradation becomes a thing of the past. After all, how do you add noise to the number 1? You see how powerful and flexible digital signal storage is. There are lots more tricks you can perform with numbers and we'll be examining some of them in due course.

Midi software and hardware

NOW you know how Midi works you can forget about it for the time being because most programs shield you from the system's intricacies. Plenty of software will take you down among the bits and bytes if you want to go that far but you can create a lot of music without knowing anything about Midi – just plug in and follow the instructions.

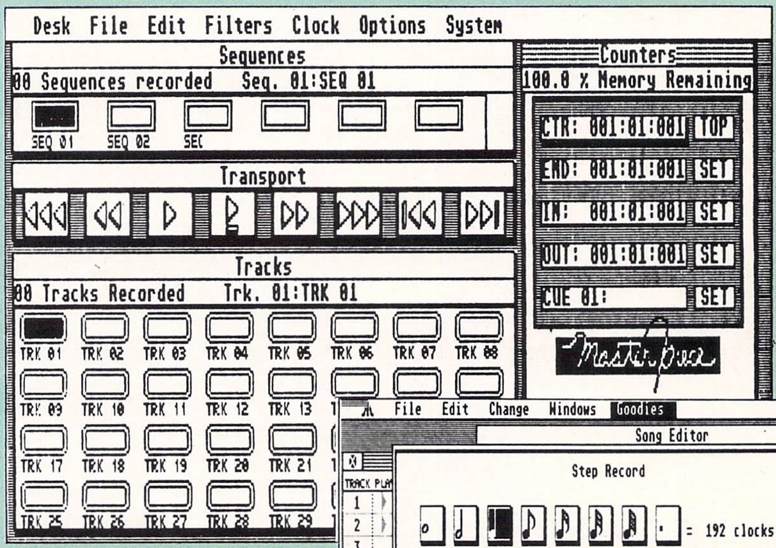
Programs which let you record and playback music are generally referred to as sequencer software – such as Sonus' Masterpiece shown here. Not so long ago a sequencer was a hardware device which could only play a fixed number of say 10 or 20 notes, hardly enough for a chorus, never mind a whole tune. But it was enough for a repetitive bass line and this type of sequencer was very popular with groups such as Tangerine Dream. Modern sequencers, however, could be more properly called digital multi-track recorders and, as we have seen, they can outperform their analogue cousins in many ways.

One of the drawbacks to digital recording, however, is that each music part requires a sound source in order to be heard. Using a tape recorder you can record several parts with one synth, changing the sounds on the synth each time you record a part. With Midi the parts are stored as numbers and the whole performance is played all at once, "live" as it were. If you record 16 parts you'll need 16 sounds sources or instruments for playback.

There are two ways around this. One involves using a multi-track tape recorder. All the music parts are recorded into the computer one at a time and a timing or synchronisation signal is recorded on to one of the tape recorder tracks.

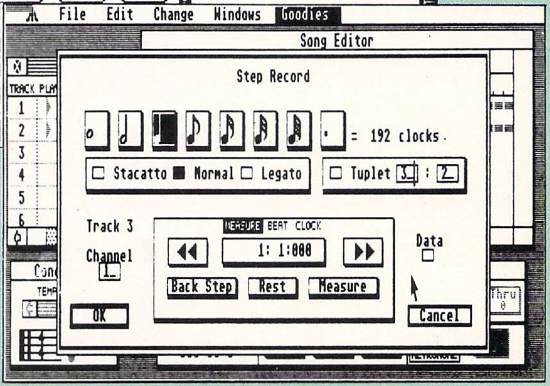
This signal is fed into the computer and used to synchronise the playback of each of the parts. They can be played one at a time, each using a different sound, and recorded this way on to the multi-track, all in perfect synchronisation.

The second method involves using a multi-timbral synthesiser. These are very interesting beasties and were developed as a direct result of Midi. You know synthesisers can produce lots and lots of sounds, well a multi-timbral instrument can play several sounds at the same time. A multi-timbral instrument may be able to play four or eight different music parts at once – under computer control, of course – and are ideal companions for Midi sequencers, adding necessary versatility.



Desk File Edit Filters Clock Options System
Sequences
00 Sequences recorded Seq. 01:SEQ 01
SEQ 01 SEQ 02
Transport
Tracks
00 Tracks Recorded Trk. 01:TRK 01
TRK 01 TRK 02 TRK 03 TRK 04 TRK 05 TRK 06 TRK 07 TRK 08
TRK 09 TRK 10 TRK 11 TRK 12 TRK 13
TRK 17 TRK 18 TRK 19 TRK 20 TRK 21
TRK 25 TRK 26 TRK 27 TRK 28 TRK 29
Counters
100.0 % Memory Remaining
CTR: 001:01:001 TOP
END: 001:01:001 SET
IN: 001:01:001 SET
OUT: 001:01:001 SET
CUE 01: SET
MasterPiece

The Main Screen from Sonus' Masterpiece



MasterPiece event editor main screen

Using Midi hardware

NOW that we have a basic understanding of Midi let's look at its role in music. Its major advantages over pre-Midi music making can be summed up in two words – power and flexibility.

The advantages of Midi

MIDI gives you the power to record and arrange an enormous number of musical parts and the flexibility to order, change and edit them. But more than that, as Midi was conceived as a standard, it enables pieces of equipment produced by different manufacturers to be connected together, thus helping stave off the ravages of obsolescence.

Before Midi, trying to connect synths, sequencers and drum machines together was a veritable nightmare – and computers just didn't have a look in. There were sometimes even problems trying to connect instruments produced by the same manufacturer. With Midi just about any piece of equipment can be connected to any other.

As Midi uses digital signals, it was just asking for a computer to be plugged into the system somewhere along the line, and it didn't take long for some clever boffins to come up with Midi software – and that's where we and our ST come in.

Midi was also responsible for the development of two new types of instrument – the keyboardless synthesiser – the expander – and the multi-timbral instrument. Let's see what's so special about them.

Expanders

MIDI makes music by sending Note On and Note Off messages. So plugging one Midi keyboard into another will allow you to control the second keyboard from the first, as shown in Figure I. The controlling keyboard is referred to as a Master or Mother keyboard, depending upon your genealogical instincts, and the other keyboard as the slave.

But haven't we got something here surplus to requirements? If you are to control the second keyboard from the first a whole set of keys is going to waste. Why buy what you aren't going to use? So was born the expander, which has all the features of a synthesiser except a keyboard. Expanders are ideal for use with your ST, especially if you aren't a keyboard player. You can program music on the ST and play it back through an expander.

Multi-timbral instruments

AS you only have one pair of hands you can only play one musical part at once. You would find it pretty difficult to play say, a piano, bass and string part at the same time. A software sequencer, however, allows you to record many parts of music, one at a time, and store them inside the ST. The process is not unlike using a multi-track tape recorder.

OK, so you've recorded three music parts in your sequencer. Now you need three synthesisers to play them back – one to produce the bass sound, one a piano and the third a string sound. This looks as if it could get pretty expensive.

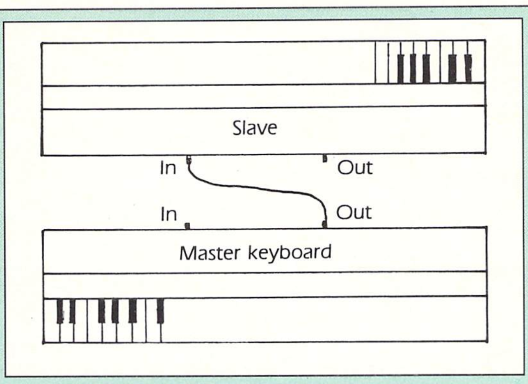


Figure I: A Master keyboard controlling a slave

But Midi comes to the rescue again with the multi-timbral synthesiser.

You know that synths can produce lots of different sounds. In the same way, a multi-timbral device can produce lots of different sounds at the same time. It's rather like having many individual synthesisers inside one unit.

It is largely the multi-timbral synthesiser which has made Midi and computer music so popular and affordable. Multi-timbral instruments capable of playing eight sounds at once are available for a few hundred pounds. Such power and versatility – and at such a price – was unheard of even a few years ago.

Midi channels

SO how can you use Midi to play several sounds at the same time? This is accomplished by transmitting different music parts on different channels. Midi supports 16 channels which means it can, theoretically, handle the same number of music lines.

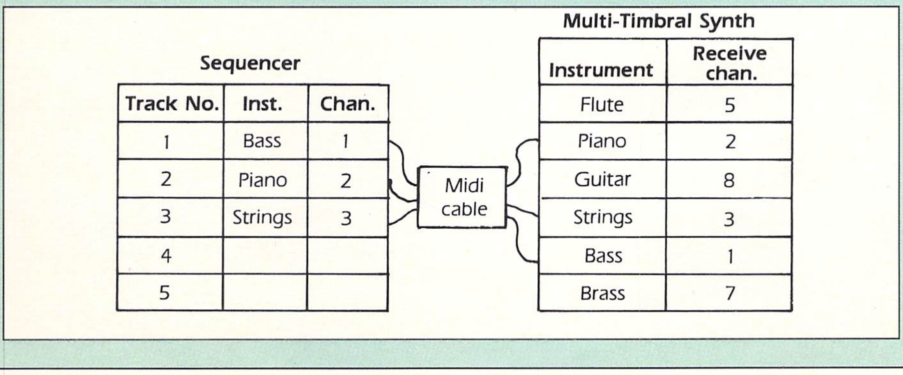
Let's take our piano, bass and drums example. Having recorded them on to separate tracks in a sequencer, we would instruct it to play back the tracks on different Midi channels, say the bass on channel one, the piano on channel two and the strings on channel three. This is a simple operation on most sequencers.

Next we would assign bass, piano and string sounds on our multi-timbral synth to their respective Midi channels. The system works rather like a 16-channel TV set. Although the synth would receive information for all three parts, the sounds would ignore messages which were not on their channel (see Figure II).

You can change the sounds played by each channel by transmitting a Patch Change message. This is sent automatically when you select a new sound on a synth's front panel.

If two synths were connected on a one to one basis, changing the sound on one will change the sound on the other. Note that this will select the same patch number on the second synth as if you had pressed the button yourself – the actual sound or the parameters which make up the sound are not sent via Midi.

Usually the ability to transmit different sequencer tracks on different channels is all that is required to produce multi-



Sequencer

Track No.	Inst.	Chan.
1	Bass	1
2	Piano	2
3	Strings	3
4		
5		

Multi-Timbral Synth

Instrument	Receive chan.
Flute	5
Piano	2
Guitar	8
Strings	3
Bass	1
Brass	7

Figure II: Using Midi channels to play a three-part recording

part music, but some sequencers also let you send patch change messages on individual channels.

The necessity for this will depend upon your particular set-up. For example, if you are using a synth which is not multi-timbral you would set it to receive on a single channel and select different sounds simply by sending different patch change numbers. Of course, it would still only be able to play one sound at a time.

Making an arrangement

IF all this theory seems a little complex, it's really quite simple in practise. Let's look at a couple of very simple examples using Activision's The Music Studio and Electronic Arts' Music Construction Set.

Both allow you to compose three-part music on the screen in traditional notation by clicking notes on to the stave. Both programs are basically designed to use the ST's sound chip but you can also play the music via Midi.

Figure III shows the Midi Parameters screen from The Music Studio. You can see the instrument names on the left followed by their Midi channel number, preset number and range. Music Studio uses different colours to distinguish different instruments and you can change instruments at will in the score.

Let's assume you've constructed a piece using lots of different instruments. To play the sax part via Midi, set its channel number to the number that the sax sound on your synth is set to receive on.

In Figure III the sax is shown as instrument three and it has been set to transmit on channel three. If you have a multi-timbral synth, assign the sax sound to channel three.

If you are using a synth which is not multi-timbral, set it to receive on channel three and change the Preset parameter in The Music Studio to correspond to the patch number of the sax sound. In Figure III this is set to 24. In this way you can make the instruments in the sequencer tie in to the sounds on your synth. The settings are saved with the score.

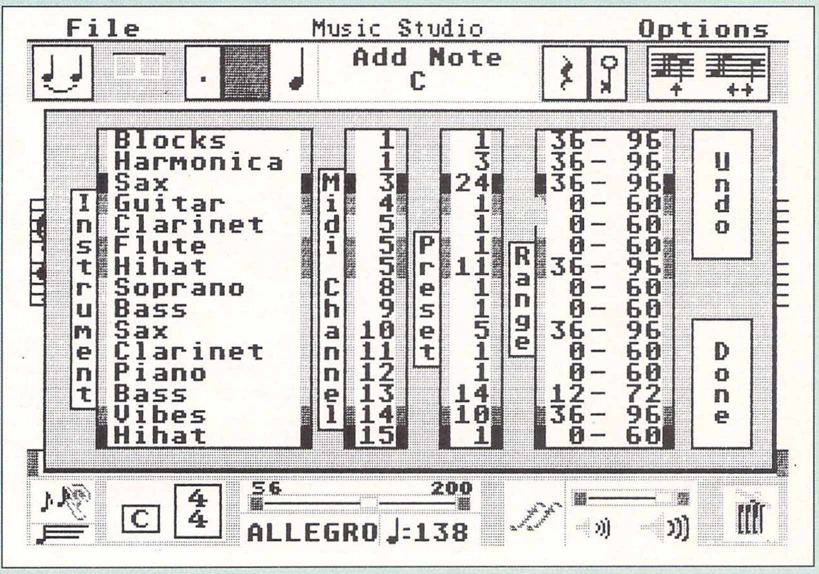
Figure IV shows the Music Construction Set's Midi Parameters screen. Here you simply assign Midi channels to the three part numbers (which it calls voices). Preset and Range values are set from another screen.

The Range values are really transpose functions. Both programs have a range of only five octaves, but Range lets you move that octave range high or lower in relation to the notes you see onscreen.

For example, to make a bass guitar play lower than the lowest note the program can display, you would simply lower its range – see how easy it is to alter pitch when the note data is a series of numbers.

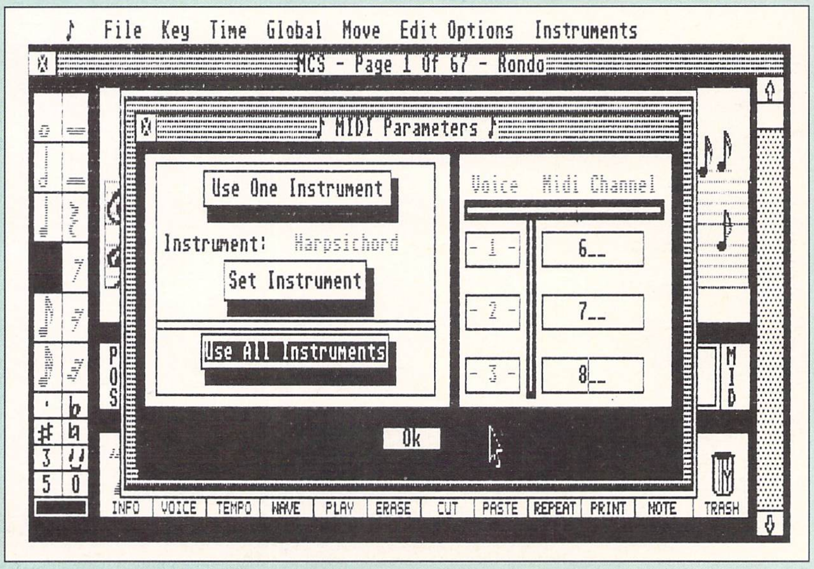
You should now have enough information about Midi to know how to use a sequencer program to change channels and sounds on a Midi instrument. Every synth and sequencer package has a slightly different method of operation, so it is important to read the manuals carefully.

You can have a lot of fun simply playing back music files via Midi and making your own Midi orchestrations and there are lots of public domain music files available for The Music Studio.



Instrument	Midi Channel	Preset	Range
Blocks	1	1	36- 96
Harmonica	1	3	36- 96
Sax	3	24	36- 96
Guitar	4	1	0- 60
Clarinet	5	1	0- 60
Flute	5	1	0- 60
Hihat	5	11	36- 96
Soprano	8	1	0- 60
Bass	9	1	0- 60
Sax	10	5	36- 96
Clarinet	11	1	0- 60
Piano	12	1	0- 60
Bass	13	14	12- 72
Vibes	14	10	36- 96
Hihat	15	1	0- 60

Figure III: The Music Studio's Midi Parameters screen.



File Key Time Global Move Edit Options Instruments
MCS - Page 1 Of 67 - Rondo
MIDI Parameters
Use One Instrument
Set Instrument
Use All Instruments
Ok
INFO VOICE TEMPO WAVE PLAY ERASE CUT PASTE REPEAT PRINT NOTE TRASH
MIDI

Instrument: Harpsichord

Voice	Midi Channel
1	6__
2	7__
3	8__

Figure IV: The Music Construction Set's Midi Parameters screen.

Getting into Midi

Entering notes

NOW that we've seen how useful Midi can be in the creation of a piece of multi-part music, let's see how we get the music into the computer in the first place. There are two methods of entering notes: Real-time and step-time. With real-time input you play a Midi keyboard and the sequencer records all the notes, program changes and so on, *live* as you play. It also records all your wrong notes and sloppy timing. However, remember that the music is stored as numbers, and with a little software help you can make corrections.

Step-time input involves entering notes one at a time. It's slower, but ultimately more accurate. It's also an ideal way of creating music if you can't play a keyboard or Midi instrument.

Most sequencers, especially the professional variety, are heavily biased towards real-time recording, although many now include good step-time facilities as well.

One of the simplest methods of step-time input, and one which many ST musicians will be familiar with, involves clicking notes on to a stave with the mouse. This is used by Activision's Music Studio, Electronic Arts' Music Construction Set, Kuma's Minstrel and other similar programs.

This kind of step-time entry is useful, for example, should you want to enter tunes from sheet music. It is also quite rewarding to see music expressed in notation rather than as a list of numbers, especially if you've written it yourself.

These programs can only handle a few aspects of music notation. You need to look at pro-level programs such as C-Lab's Notator if you want more control and flexibility.

Editing

ONE of the benefits of ST-based sequencing is the amount of control it offers over your music once it has been recorded. Most sequencers have editing facilities which let you alter whole sections of the piece or just individual notes.

Many editors show the notes as a series of numbers which indicate the bar and beat at which the note begins, its velocity and duration. This method was used even before the ST became the music machine of the micro world.

It was taken up by many programs and is still a popular method of editing. However, it is very numeric and some programs – such as Steinberg's Pro-24 and Trackman – as shown in Figure I – have adopted a grid edit system which many musicians find easier to use.

At the highest level you will find programs such as C-lab's Notator and Steinberg's Pro-24 – seen in Figure II – and MasterScore which show music data in traditional notation.

Many musicians, especially if they are not fully conversant with traditional notation, prefer other methods of editing. Some sequencers also include a drum grid edit page in which you can create drum patterns.

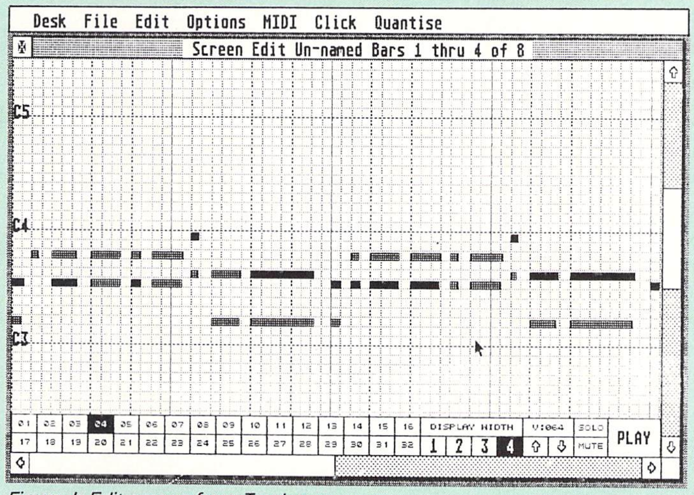

Desk File Edit Options MIDI Click Quantise
Screen Edit Un-named Bars 1 thru 4 of 8
C5
C4
C3
DISPLAY WIDTH
V:064
SOLO
MUTE
PLAY

Figure I: Edit screen from Trackman

Quantisation

NOW let's see how software can tidy up real-time note entry. You will find this in your sequencer under the heading of quantisation or auto note correct.

In order to keep track of the timing of each note, sequencers use a clock which ticks away during recording and playback. The speed or resolution of the sequencer clock is generally fixed and is expressed in terms of pulses per quarter note – abbreviated to ppq or ppqn. It is typically 96 ppq, although it can be higher or lower. Note that this is an internal timer, and has nothing to do with tempo.

The clock reads each note as it is played and places it on to one of its clicks or pulses. Four semiquavers for example, would be placed on clicks 1, 7, 13 and 19. Well, that's the ideal, but if your timing is a little sloppy the notes may end up on, say, clicks 2, 6, 14 and 21.

The quantisation process looks at the resolution you wish to correct to – in this case it would be 16th notes – and then pulls or pushes the notes on to the nearest relevant click. This is done automatically – you just have to select the resolution.

Quantisation can help tidy up poor timing, but it can also result in mechanical runs of notes. Some sequencers, therefore, have a *human quantise* function which will only correct notes if they are out by a large amount. Computers do not have to produce robotic music unless you want them to.

Tracks and channels

YOU have probably heard of sequencers with 24, 32, 64 or more tracks. Before we go any further, let's make clear the difference between a track and a channel. We've already looked at Midi channels, particularly in relation to multi-timbral instruments, so let's

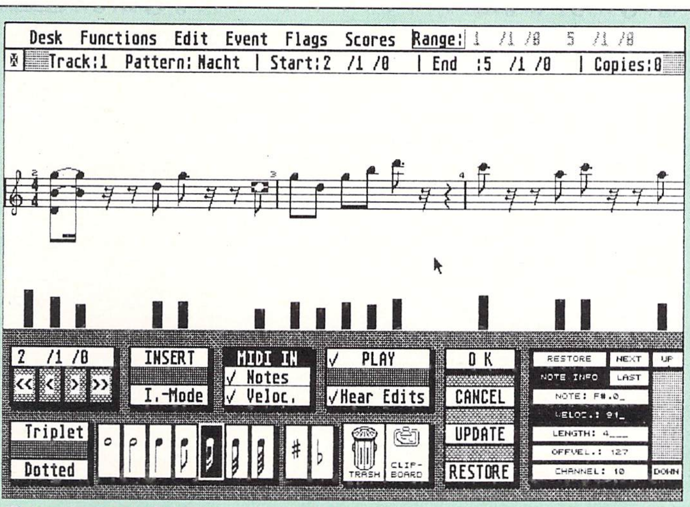
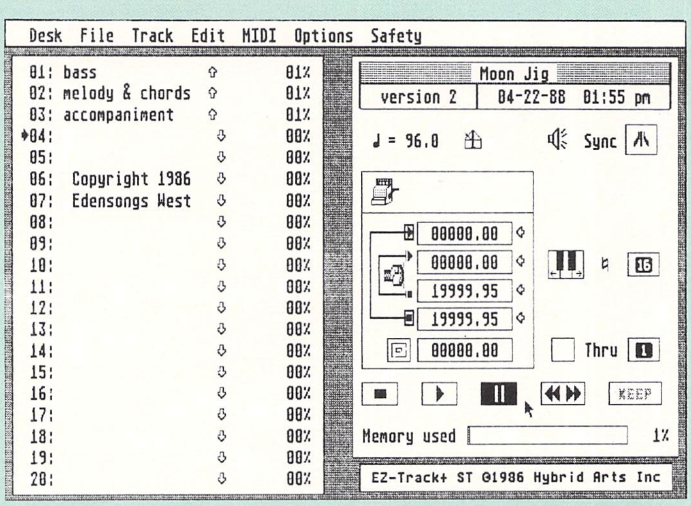


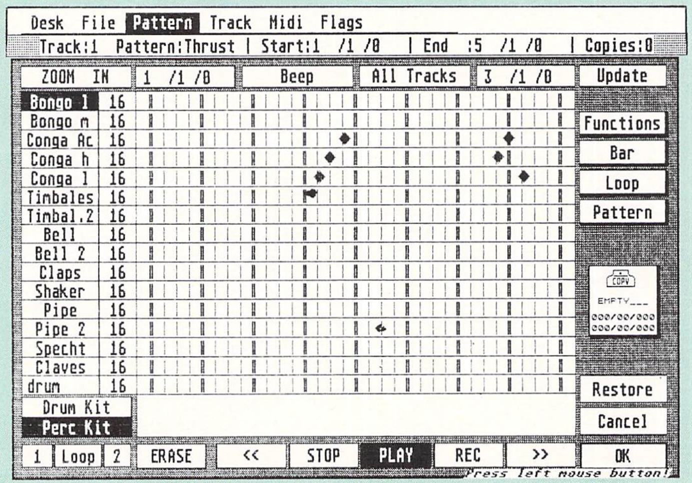
Figure II: Score-Edit screen in Pro-24

Getting into Midi


Desk File Track Edit MIDI Options Safety
Moon Jig
version 2
04-22-88 01:55 pm
♩ = 96.0
Sync
Thru
KEEP
Memory used
1%
EZ-Track+ ST ©1986 Hybrid Arts Inc

```
01: bass               01%
02: melody & chords    01%
03: accompaniment      01%
04:                    00%
05:                    00%
06: Copyright 1986     00%
07: Edensongs West     00%
08:                    00%
09:                    00%
10:                    00%
11:                    00%
12:                    00%
13:                    00%
14:                    00%
15:                    00%
16:                    00%
17:                    00%
18:                    00%
19:                    00%
20:                    00%

00000.00
00000.00
19999.95
19999.95
00000.00
```

Figure III: Main screen of EZ-Track Plus showing tape recorder controls



```
Bongo 1    16
Bongo m    16
Conga Ac   16
Conga h    16
Conga l    16
Timbales   16
Timbal.2   16
Bell       16
Bell 2     16
Claps      16
Shaker     16
Pipe       16
Pipe 2     16
Specht     16
Claves     16
drum       16
```

Figure IV: Drum-Edit screen in Pro-24

see what tracks are and how they are used in music composition.

Early software sequencers used the audio multi-track tape recorder as a model for their method of operation – for comparison, audio multi-track recorders have 4, 8, 16 or 24 tracks.

Musicians were used to tape recorders and by mimicking their features, software sequencers became a natural extension of the multi-track. In fact many current sequencers have Play, Record, Fast Forward and Rewind controls – as shown in Figure III.

The concept of a track was carried over from multi-track recorders, too. Typically, each instrument would be recorded on to a different track on the audio tape, although a drum kit, for example, would often be recorded across several tracks.

Early software developers found it convenient to use the track concept, and one of the simplest ways of creating music with Midi is to record a different instrument on each track and give each a separate channel number. In fact some sequencers let you allocate a specific Midi channel to each track so that anything you record on it plays back over that channel.

Other programs take a different approach and play back a track on the channel on which it was recorded. The first method is possibly more useful, especially for the beginner, as it helps you think in terms of instrument parts – which is how music is generally constructed.

It also allows you to record everything from one Master keyboard and play it back on the correct Midi channels via a multi-timbral expander.

But the multi-track recording concept is not the only one. Some software allows you to record a number of musical phrases which are not immediately associated with any track. You are free to place them anywhere in the piece, on any track and have them play back over any Midi channel.

Bouncing and mixing

MOST sequencers allow you to bounce or mix tracks together. You can also do this with audio recorders but how many audio recorders let you un-mix a track?

With software, it's simply a matter of looking for data recorded on different Midi channels and putting it on different tracks. This is in addition to other methods of digital data manipulation which allow you to speed up, slow down and transpose your music.

Note then, that tracks and channels are not the same, although in some software sequencers they can be closely linked. It is quite possible, for instance, for one track to contain information relating to all 16 Midi channels.

Making an arrangement

WITH this information under our belt we can proceed to arrange a piece of music. Don't think you have to record a piece from beginning to end. Look at your sequencer's edit facilities, such as copy and append tracks, to see how you can record the music in sections.

Some musicians like to record a complete single track – say bass or drums – and use it as a foundation on which to add chords and solo parts. Others prefer to record the complete work in sections, making sure each is correct before moving on to the next.

The method for you will depend upon the piece you want to record, your sequencer – and your temperament.

It can often be useful to record a single part across several tracks, especially if there are difficult bits which you are not guaranteed to be able to play correctly first time through. When all the tracks sound right playing together, mix them into one track.

Recording drums

MANY musicians now record drum tracks into their sequencer in the same way as they record other music parts.

If you look in your drum machine's manual you will see that each drum sound has a note associated with it. For example, on the Roland TR-505 playing the D above Middle C will trigger the High Cowbell.

You can usually alter the key number assignment allowing you to map a comfortable drum pad layout on to your keyboard. You can then create a drum track by playing the keyboard in real-time. This is likely to produce a more natural drum track, but you can always pull beats into line with the quantisation feature – Steinberg's Pro-24 drum edit screen is shown in Figure IV.

One of the benefits of this method of drum track creation is that the track is stored with your song so you don't have to worry about overwriting the patterns stored in your drum machine.

Experiment with your sequencer until you become familiar with its use of tracks and channels. Then when you know how to use your Midi equipment to put different sounds on different channels you will be able to create musical arrangements of your own.