STOS

The Game Creator

USER GUIDE

MANDARIN SOFTWARE

ATARI ST

STOS
The Game Creator

JAWX

STOS Basic was developed by:

François Lionet	*STOS Programmer*
Constantin Sotiropoulos	*STOS Programmer*
Frédéric Pinelet	*STOS Designer*
Jacques Fleurance	*STOS Marketing*

MANDARIN

UK design and marketing:

Chris Payne	*Marketing Manager*
Stephen Hill	*Manual Author*
Alan McLachlan	*Manual Editor*
Richard Vanner	*Project Coordinator*
David McLachlan	*Programming/Graphics*

STOS packaging by Ellis, Ives and Sprowell Partnership, Wakefield

If you have any difficulty with this product, please write to:

Mandarin Software
Europa House, Adlington Park
Adlington, Macclesfield SK10 4NP

ISBN 0 948104 99 6

Contents

Appendices

1 Introduction

CONGRATULATIONS on buying **STOS – The Game Creator**. This exciting package hits a new high in software standards, giving you the ability to design and create arcade-style games faster and easier than ever before.

The package is based around STOS Basic, an incredibly powerful new language with a staggering 320 commands – many of which have more than one use.

A feature that makes STOS Basic stand out is that it is not a Gem-based language. This allows it to run much faster than any other Basic on the ST and also takes away many of the restraints caused by the use of Gem, such as only being able to use one resolution.

STOS Basic replaces these Gem functions with ones of its own. There are powerful windowing facilities and easy-to-use file selectors – and drop down menus are simple to create.

Supplied with the STOS Basic disc are two other discs containing the accessories and games. The accessories are what makes STOS really come to life, including specially-designed programs which work alongside your own program to help speed up development. The list of accessories include a Sprite Definer, Music Editor, Character Editor, Icon Editor and many more.

The games disc contains three written in STOS Basic – Bullet Train incorporating fast horizontal-scrolling, Zoltar, a Galaxian-style shoot-'em-up which was written in just three days, and Orbit, a feature-packed bat-and-ball game.

As you can see, STOS is not just another Basic – it's a full-blown developers' kit which can be used by people of any age and experience. STOS also has an exciting future and there are plans for a number of extension discs containing new commands.

Whatever your knowledge of progamming, STOS has something to offer you. If you have never written a game before, the prospect of creating your first game may be quite daunting. But do bear in mind that many of the all-time classics like Confuzion, Zenji, Tetris and Split Personalities – to name but a few are – uncomplicated programs with one or two features which have entertained computer owners in their thousands. The strength of your game will mainly be based on your ideas, and not just your programming skill.

Making a back-up

The STOS discs are not protected, which means that you can make back- ups or upload the discs on to a hard drive if you have one. But please don't give copies to other people. STOS took two years of intense programming to reach its current state, so the higher the sales, the greater will be our incentive to develop new extension discs and accessory programs.

The three discs supplied are your STOS master discs and must be looked after. You should copy each one on to a new, formatted disc and place the original master discs safely out of harm's way. So if your discs get damaged, corrupted or even have files deleted from them, you can go back to the master disc to produce new working copies.

The procedure for making back-ups is as follows:

1 Boot up the Gem Desktop.

2 Place a blank disc into drive A and format it using the menu command.

3 Now place the master disc into drive A and drag the drive A icon on to the drive B icon.

4 Follow the instructions displayed in the dialogue boxes.
5 Repeat actions 2 to 4 for the other two discs. Refer to your Atari ST manuals if you have trouble copying the discs.
6 Once the copy is complete, store the master discs in a safe place.

Run-time programs

When you have written a program in STOS Basic you may wish to get it published as a commercial game. This is no problem in STOS – all you have to do is save your program with a .PRG extension to create a copy which can be booted from the Gem Desktop, but please ensure that you mention that you have used STOS on the loading screen. For more technical information about this subject see Chapter 3 and Appendix B.

We at Mandarin Software are very keen to publish games written using STOS. Address your correspondence for the attention of the Software Manager, Mandarin Software, Adlington Park, Adlington, Macclesfield SK10 4NP. If you decide to protect your game may we suggest that you allow other STOS users to examine and modify your sprite and music banks? This way your game will be of greater interest to STOS owners and could ensure higher sales.

We want to build up a vast database of STOS users so that you can benefit from the input of people all around the country. To help us do this we would urge you to fill in the registration form enclosed in the STOS packaging so we can find out what users want. You also stand to win a prize in our monthly draw.

Using this manual

We have dedicated most of the manual around the special functions offered by STOS Basic. If you have no prior knowledge of Basic, you will need to purchase an introductory text such as Alcock's Illustrating Basic (Cambridge University Press). In our opinion, this book gives you an excellent insight into programming Basic. We still feel you can easily pick up Basic from this manual, but various techniques would not seem apparent if you learnt it this way.

The STOS manual is set out in a tutorial fashion, giving you many examples of how to use each instruction. Programs have been included to emphasise how certain instructions can be used to their full effect. There is also a comprehensive appendix which will explain various technical information to experienced programmers.

When you enter the example programs listed in the manual you must remember that most of them are designed to work in the low resolution mode on colour monitors, as most commercial games use this mode. However STOS Basic can operate in all three modes, which means that owners with monochrome monitors can use the language.

One last point. Try to get into the habit of booting STOS directly from disc rather than from Gem. This will free 32k of extra memory for you to use in your programs.

2 Guided Tour

STOS Basic has to be one of the most powerful versions of Basic which has ever been written for the Atari ST. It includes a wide range of facilities for sprite manipulation, screen flipping, and the generation of high quality music. It is also easily expandable, so you won't be left behind by any future developments.

The quality of STOS Basic as a development system has to be determined by the quality of the programs which can be produced with it. To provide an example of what you can achieve with this package, we have supplied you with three games written entirely using STOS Basic. These can be found on the games disc, and can be listed and amended like any other Basic program. Although STOS Basic may seem very games oriented, there are a number of other possible applications – such as educational software – for which it would also be ideally suited.

In this chapter we'll be giving you a guided tour of just some of STOS Basic's exceptional features. But first, a plea from the heart. If you have not already made a backup of this package, jump immediately to the section on MAKING A BACKUP. Although we at Mandarin will be happy to replace your disc for a nominal handling charge if something goes wrong, you will be deprived of STOS Basic while it's being re-duplicated.

The sprites

We'll start our tour with a brief look at the STOS Basic sprite commands. These allow you to move and animate a sprite using simple, easy-to-understand Basic instructions. There is no poking around in the ST's memory, and you don't need to know anything about the ST's internal workings in order to use them.

Furthermore, STOS Basic comes complete with an excellent Sprite Editor which can be installed permanently in your ST's memory, and then entered at any time by pressing just two keys. This enables you to design, test, and modify your sprites in one smooth operation.

Let's have a look at the sprite commands in action. Before we can use these instructions, we will first need to load some example sprites from the Accessory disc. Place the disc into your drive and type in the line:

```
load "animals.mbk"
```

We can now display one of these sprites on the screen using the SPRITE command:

```
sprite 1,100,100,1
```

Similarly we can examine the rest of the sprites by typing in the following and pressing any key to view each sprite in turn:

```
for A=1 to 50:sprite 1,100,100,A:wait key:next A
```

Up to 15 of these sprites can be placed on the screen at any one time. As an example, enter the line:

```
for A=1 to 15:sprite A,1,A*10:wait key:next A
```

Moving a sprite

Now for some movement!

We first draw sprite number 1 on the screen with:

```
sprite 1,10,100,1
```

This displays a sprite looking rather like an octopus. This was generated in a matter of minutes using the STOS Basic Sprite Editor.

Let's add a little movement to this sprite:

```
move x 1,"(1,1,300)L"
move on
```

The octopus is now moving smoothly across the screen in the X direction. Since these sprite movements are performed using interrupts, they are therefore able to execute completely independently of your Basic program. We can prove this by typing in the following line:

```
for A=1 to 10000:P=P+1:next A:print P
```

As you can see, the octopus continued onwards, at the same time STOS Basic was busy executing the FOR...NEXT loop.

So far, we have only moved our sprite in a simple straight line. We can however, easily specify a whole list of these movements in exactly the same way.

```
sprite 1,0,100,1
move x 1,"(1,3,100)(1,-3,100)L"
move on
```

When you type in the above commands, the octopus now walks slowly back and forth along the screen.

The last few examples were restricted to horizontal motions. But there's also a separate MOVE Y instruction to move the sprite up and down as well. To see how this works, enter the lines:

```
move y 1,"(1,3,30)(1,-3,30)L"
move on
```

Finally, we can combine any sequence of horizontal and vertical motions like so:

```
sprite 1,0,0,2
move x 1,"(1,2,150)(1,-10,30)L"
move y 1,"(3,1,100)(3,-1,100)L"
move on
```

This technique can be used to rush all 15 sprites across the screen in any direction. Look at the game Zoltar for an impressive demonstration of the speed of these commands.

Animation

Each of these sprites can be animated automatically with a special ANIM

instruction. ANIM displays a list of sprite images on the screen, one after another. As this feature is performed using interrupts, it can be combined with MOVE to produce some very effective animation.

Type in the following small example:

```
sprite 1,100,100,1
box 100,100, 32,132
anim 1,"(1,10)(2,10)(3,10)(4,10)L"
anim on
```

The octopus is now waving its arms about frantically. This is probably because it's trapped in the box. Let's put it out of its misery and release it, using the MOVE commands like so:

```
move x 1,"(1,4,75)(1,-4,75)L"
move y 1,"(1,4,24)(1,-4,24)L"
move on
```

Freedom at last! Our octopus has escaped.

It is important to realise that, like all the sprite commands, ANIM causes no delay to your current program. For a further example of animation, see the program on page 7.

The STOS Basic sprite commands

SPRITE	Draw a sprite
MOVE	Start/stop movements
MOVE X	Move sprites left and right using interrupts
MOVE Y	Move sprites up and down
ANIM	Animate a sprite
PUT SPRITE	Copy a sprite to the screen
GET SPRITE	Make a rectangular section of the screen into a sprite
UPDATE	Update sprites
AUTOBACK	Switch off link between sprite background and real screen
X SPRITE	Get X coordinate of a sprite
Y SPRITE	Get Y coordinate of a sprite
MOVON	Check if sprite currently in motion
COLLIDE	Test of sprite collisions
LIMIT SPRITE	Limit sprite movements to only part of a screen
ZONE	Test if sprite enters a rectangular section on the screen
SET ZONE	Define one of 128 rectangular zones
RESET ZONE	Clear current zones
PRIORITY	Change sprite priority
REDRAW	Redraw sprite
DETECT	Detect pixel under sprite
SYNCHRO	Synchronise sprite with scrolling background

A complete description of these instructions can be found in Chapter 4

Manipulating the screen

If you thought the sprite commands were impressive, wait until you see the screen manipulation routines! STOS Basic has the ability to scroll, move and copy parts of the screen. Put the system disc into the drive and type:

```
load "\stos\pic.pi1"
```

This loads the title picture from the STOS Basic folder into the current screen. One minor snag with these screens is that they each take up over 32k of space on the disc. Fortunately STOS Basic includes a powerful Screen Compactor accessory which can cram any screen down to as little as 7k. An example screen in this format has been placed on the accessory disc in the file BACKGRD.MBK. Let's load it into the ST's memory:

```
load "backgrnd.mbk"
```

The above command loaded the screen into one of STOS Basics 16 memory banks (See Chapter 3). We can now unpack it using the UNPACK command like this:

```
unpack 11,physic
```

The effect of the above instruction was to expand the picture into the current screen. If you now move the mouse, the picture will be steadily erased. This is because STOS uses a separate background screen for the sprites. Also note that the image seems to be flashing. When STOS Basic is first loaded, colour number 2 is initially started flashing. See FLASH for more details. You can turn off this feature using:

```
flash off
```

Let's see what happens when we copy the picture into the sprite background instead.

```
flash off
unpack 11,back
```

If you move the mouse around on the screen as before, the picture will now be progressively drawn.

We can incorporate these instructions into a small STOS Basic program.

Example:

```
10 cls:flash off:unpack 11,back
20 appear back,rnd(78)+1
30 wait key:goto 10
```

In this example we've introduced an interesting new instruction called APPEAR. This command fades between two screens using one of 79 possible effects.

Here's another example, using the FADE instruction:

```
10 mode 0
20 fade 3
```

```
30 reserve as screen 15
40 load "\stos\pic.pi1",15
50 fade 25 to 15
60 appear 15
```

Now for something rather different. One of the most impressive features of STOS Basic is its ability to change the size of any image displayed on the screen. To that end it provides you with the two instructions REDUCE and ZOOM.

We can demonstrate the REDUCE command by adding the following line to the program above.

```
70 reduce physic to 200,50,280,100
```

This reduced the entire screen to a quarter of its normal size and copied it to the rectangle starting at 200,50.

As you might expect, the ZOOM command has the opposite effect, and magnifies a section of the screen. We can see the effect of one of these instructions by entering the lines:

```
mode 0:locate 0,0 : print "STOS Basic"
zoom physic,0,32,88,40 to 0,40,319,198
```

This prints the string **STOS Basic**, and then expands to fill the screen.

An equally important capability of STOS Basic is to enable you to copy large sections of the screen from one place to another at high speed. This can be achieved using a powerful SCREEN COPY function. We can incorporate an example of this instruction into our program simply by inserting a new line at 40:

```
80 screen copy physic,200,50,280,100 to physic,100,50
```

This places a copy of the miniature screen generated with REDUCE at the coordinates 100,50

Finally, a few words about the screen scrolling commands. These allow you to scroll any part of the screen either vertically or horizontally. We can demonstrate these instructions by inserting the lines below:

```
80 def scroll 1,50,90 to 250,110,1,0
90 def scroll 2,140,10 to 160,190,0,1
100 scroll 1 : scroll 2 : goto 100
```

Now for an example which combines sprites and screens into a single program. Put the accessory disc into the drive and type:

```
load "backgrnd.mbk",11
load "animal":rem Loads the sprites

10 mode 0 : flash off
20 unpack 11,back : appear back,30
30 reduce physic to 200,50,280,100
40 sprite 1,130,80,80
50 move x 1,"320(2,-6,0)L"
60 anim 1,"(5,5)(6,5)(7,5)(8,5)(9,5)(10,5)(11,5)(12,5)1"
70 move on : anim on : wait key
```

The screen manipulation commands	
APPEAR	Fade between two screens using a pattern
FADE	Fade the present colour palette in single steps to a new setting
BACK	Return the address of the sprite background
PHYSIC	Return the address of the physical screen
LOGIC	Return the address of the logical screen
DEFAULT	Return default addresses
REDUCE	Reduce the screen in size
ZOOM	Expand the screen in size
SCREEN COPY	Copy a section of the screen from place to place
SCREEN SWAP	Swap physical screen with logical screen
SCREEN$	Assign part or all of a screen to a string
DEF SCROLL	Define a scrolling zone
SCROLL	Scroll part of the screen
GET PALETTE	Load the colours of a screen in memory into physical screen
CLS	Clear part or all of screen
WAIT VBL	Wait for next vertical blank
UNPACK	Unpack a screen in compressed format
PACK	Compact a screen to save memory
See Chapter 7 for a full explanation of the screen instructions.	

General graphics

STOS Basic supports a number of the more normal graphics operations such as CIRCLE, BOX, and POLYGON. One major difference between STOS and other Basics however, is its ability to change the graphics resolution at any time during a program, using just a single STOS Basic instruction.

Example:

```
10 mode 0:print "Low resolution"
20 print "Press a key to change graphics modes"
30 wait key:mode 1
40 print "Medium resolution"
```

Note that for obvious reasons the MODE command has no effect whatsoever on monochrome only systems.

Another interesting command is SHIFT which rotates the screen pallete through every possible colour combination. To demonstrate the effect of the SHIFT instruction type:

```
shift 100
```

As you can see, the screen colours are continuosly changed every few seconds. We can turn SHIFT off with a simple:

```
shift off
```

We've saved the best till last. This is the FLASH instruction which allows you to animate any colour through a sequence of up to 16 different colour changes. Since FLASH uses interrupts, it will occur simultaneously with the rest of your program without affecting it in the slightest. Let's animate colour number 0 with the line:

```
flash 1,"(000,5)(333,5)(666,5)(777,5)(555,5)(222,5)
```

This produces a startling set of multicoloured characters.

The GRAPHICS instructions

POLYMARK	Print marker
ARC	Draw a circular arc
EARC	Draw an elliptical arc
PLOT	Plot a point
POINT	Determine the colour of a point
DRAW	Draw a straight line
BOX	Draw a hollow box
RBOX	Draw rounded hollow box
POLYLINE	Draw a hollow polygon
PIE	Draw a pie chart
EPIE	Draw an elliptical pie chart
CIRCLE	Draw a filled circle
ELLIPSE	Draw a filled ellipse
BAR	Draw a filled bar
RBAR	Draw rounded filled bar
POLYGON	Draw a filled polygon
PAINT	Contour fill
MODE	Change graphics mode
FLASH	Set flash sequence
SHIFT	Rotate colours
INK	Set ink colour
PALETTE	Set all colour assignments
COLOUR	Read/write one colour value
GR WRITING	Set writing mode
SET LINE	Set line type
SET MARK	Set marker type
SET PAINT	Set fill Type
SET PATTERN	Set user-defined fill pattern
CLIP	Set clipping rectangle
DIVX	Width of mono screen/width of current screen
DIVY	Height of mono screen/height of current screen
CLS	Clear entire screen

See Chapter 6 for a complete explanation of these instructions.

The mouse

In many respects the STOS Basic mouse pointer is rather unusual. The most obvious difference is that it is much more colourful than the one you are used to. This is largely because this pointer is really just a specialised version of a sprite. The major advantage of this approach is that you can easily set the shape of the mouse pointer to anything else you like using the CHANGE MOUSE command.

Examples:

```
change mouse 2:rem Change mouse to hand
change mouse 3:rem Change mouse to clock
```

You can also use the instruction to change the mouse into any one of the sprite images currently held in the ST's memory. We'll now demonstrate this process.

Place the accessories disc in the drive and load some sprites with:

```
load "sprdemo.mbk"
```

Now change the mouse to the first of these sprites with:

```
change mouse 4
```

and to the second with:

```
change mouse 5
```

As you can see, the number used in the above instruction is just the image number plus four.

Detecting collisions between a sprite and the mouse is easy. You can also test a specific area to the screen to see if the user has entered it with the mouse.

Reading the mouse is equally straightforward, as the position of the pointer is instantly returned by the X MOUSE and Y MOUSE functions.

Example:

```
10 locate 0,0:print x mouse,y mouse:goto 10
```

If you run this program and move the mouse across the screen, its location will be continually displayed.

The mouse commands

X MOUSE	Return X coordinate of mouse
Y MOUSE	Return Y coordinate of mouse
MOUSE KEY	Test mouse buttons
ZONE	See if mouse is in a rectangular zone
SET ZONE	Define zone to be tested
RESET ZONE	Clear zone definitions
CHANGE MOUSE	Change mouse picture
HIDE	Remove mouse from screen
SHOW	Return mouse to normal

More details of these instructions can be found in Chapter 4.

The joystick

STOS Basic includes a number of simple commands which enable you to test the movements of a joystick. Place a joystick into the right socket and type:

```
10 if jleft then print "LEFT"
20 if jright then print "RIGHT"
```

```
30 if jup then print "UP"
40 if jdown then print "DOWN"
50 if fire then boom 60 goto 10
```

The joystick commands	
JOY	Read joystick and test all functions
JLEFT	True if joystick moved left
JRIGHT	True if joystick moved right
JUP	True if joystick moved up
JDOWN	True if joystick moved down
See Chapter 4 for more information.	

Sound and music

In the bad old days of computing, you were lucky to find the inclusion of a humble BEEP instruction. The STOS Basic programmer has a much easier time of it. Not only can you produce high quality soundtracks for your games, but you can also generate a vast range of other special effects. Furthermore, if you're already an expert on the subject, STOS gives you complete control over the ST's sound chip.

Creating a piece of music couldn't be easier, as a superb Music Editor is included for your use as part of the STOS Basic package. Like the Sprite Editor, this can be loaded into memory, and called at any time straight from the keyboard. As an example, we've placed a piece of music for you on the accessory disc. Load this with the line:

```
load "music.mbk"
```

You can now play the music by typing:

```
music 2
```

This music plays independently of the rest of the STOS system in a similar way to the Sprite commands.

Let's change the speed of the music with TEMPO:

```
tempo 10
```

which slows the tune down to a crawl. Now type:

```
tempo 100
```

Fast enough for you? We can also change the pitch of the music. First the music back to normal with:

```
tempo 40
```

Now type:

```
transpose 30:rem Increases the pitch
```

and

```
transpose -20:rem Lowers the pitch
```

Finally, turn the music off using:

```
music off
```

Further examples of music can be found in Bullet Train.

Sound effects

STOS Basic also supports a number of useful functions for the production of more basic noises. The simplest of these are the SHOOT, BOOM and BELL commands. Here are a few examples for you to type in.

```
for A=1 to 10:boom:wait 5: next A
shoot
bell
```

In addition to the pre-defined effects, you can utilise the noise generator in conjuction with the ENVEL command to produce a range of more exotic sounds.

Examples:

```
click off
volume 16
noise 1
envel 10,100:Rem Aeroplane
envel 10,1000:Rem Helicopter

envel 1,1:rem Reset envelope
envel 14,80 play 14,80

envel 8,40
play 37,40
```

STOS Basic sound commands

MUSIC	Play music defined using music editor accessory
VOICE	Activate/Deactivate individual voice
TEMPO	Change speed of music
TRANSPOSE	Change pitch of music
VOLUME	Set volume of noise
ENVEL	Choose shape of note/noise
PLAY	Play a single note on one of three voices
NOISE	Generate some noise
BOOM	Make a BOOM sound
BELL	Make a BELL sound
SHOOT	Make a SHOOT sound
PSG	Access sound chip. **Warning:** Handle with care!

See Chapter 5 for more details of these commands.

Displaying text on the screen

If you've used Gem, you'll probably already be familiar with the idea of windows. Although STOS Basic is not Gem-based, it does incorporate a range of impressive

windowing operations. These allow you to create a window with one of 16 different borders anywhere on the ST's screen. Each window can have its own unique character set which can be stored in a special memory bank along with your program. Here's a simple example of a STOS Basic window:

```
windopen 1,3,3,30,10,12
```

We can delete this window with the line:

```
windel 1
```

Now for a larger example which displays 10 windows on the screen at once.

```
for i=1 to 10:windopen i,3*i,i,10,10,i:next i
```

After this line has executed, the text cursor will be placed in the last window we have defined. We can switch the cursor to another window using the WINDOW command like so:

```
window 1
window 4
window 7
window 10
```

Since we don't need these windows any more, we can delete them from the system using the DEFAULT command:

```
default
```

We'll now create a small program which displays four different character sets on the screen at one time.

First insert the accessory disc into the drive and load the fonts into memory with the lines:

```
load "font1.mbk"
load "font2.mbk"
load "font3.mbk"
```

You should then type in the following small program.

```
10 windopen 1,0,0,9,4,4,3:rem One of 3 system sets
20 windopen 2,10,0,9,4,4,4:rem First new set
30 windopen 3,20,0,9,4,4,5:rem Second new set
40 windopen 4,30,0,9,4,4,6:rem Third new set
50 input "Window ";W
60 window w:goto 50
```

Any of these sets can be used to replace the three system fonts stored on the STOS system disc. Just to make things simple, STOS Basic also supplies you with a useful Font Definer accessory which can be used to generate any new character sets you require.

In addition to the normal characters, STOS Basic includes support for special 16x16 characters called Icons. These can be displayed on the screen using the ICON$ command, or incorporated directly into menus. We have provided you with a useful set of examples in the file ICON.MBK on the accessory disc. These can be printed out using the program below.

```
new
```

```
load "ICON.MBK"
10 for X=0 to 19
20 for Y=0 to 4
30 locate X*2,Y*2
40 print icon$ (X*5+Y+1)
50 next Y
60 next X
```

Note that just as with the character sets, there's also a Icon definer to allow you to create your icons.

STOS Basic text commands

BORDER	Change window border
CDOWN	Move cursor down
CUP	Move cursor up
CLEFT	Move cursor left
CRIGHT	Move cursor right
CLW	Clear window
CURS	Hide/show text cursor
SET CURS	Set cursor type
DEFAULT	Reset windows
HOME	Cursor home
ICON$	Print an icon at current cursor postion
INVERSE	Inverse text
UNDER	Underlined text
SHADE	Shaded text
LOCATE	Set printing position
PAPER	Set text background colour
PEN	Set text colour
PRINT	Print text
USING	Formatted text
CENTRE	Print centred text
QWINDOW	Quick window activation
WINDOW	Activate a window
WINDON	Test a window to see if it's active
WINDMOVE	Move a window
WINDCOPY	Copy a window
WINDEL	Delete a window
SCRN	Get character under cursor
TITLE	Set Window title
SQUARE	Print square using text coords
XCURS	Return X coordinate of cursor)
YCURS	Return Y coordinate of cursor)
XTEXT	Convert graphic coord to text coord
YTEXT	Convert graphic coord to text coord
XGRAPHIC	Convert text coord to graphic coord
YGRAPHIC	Convert text coord to graphic coord

More details of these instructions can be found in Chapter 8.

Pull-down menus

As we near the end of our tour, we'll give you a brief glimpse at the incredibly useful STOS Basic menu commands. These enable you to effortlessly create menus

which will then work automatically using interrupts. STOS menus may be composed of either text or icons. Here is a simple example.

```
10 menu$(1)="Menu "
20 menu$(1,1)="Item1"
30 menu$(1,2)="Item2"
40 menu$(1,3)="Item3"
50 menu on
60 A=mnselect: if A<>0 then print "You chose Item number",A
70 goto 60
```

STOS Basic menu commands	
MENU ON	Start menu
MENU OFF	Halt menu
MENU FREEZE	Temporarily stop menu
MENU$(X)	Set menu title
MENU$(X,Y)	Set menu item
ON MENU GOTO	Automatic menu selection
ON MENU ON/OFF	Activate/deactivate automatic selection
MNBAR	Menu bar selected
MNSELECT	Item selected
More details of these instructions can be found in Chapter 9.	

So far we've only demonstrated a fraction of STOS Basic's capabilities. As you can see, STOS Basic provides you with everything you need to create superb games and effective educational software. The following chapters include a full explanation of all the various commands. The rest is up to you.

3 The Editor

On loading the STOS Basic package you are initially presented with a display consisting of two separate windows.

The Editor window

The Editor window is the part of the screen reserved for creating and manipulating your programs. STOS Basic supports a powerful screen editor which allows you to alter your program listings directly from the screen. The heart of this system is the text cursor which indicates the position of the next character to be input. It also marks the current line. This line can be entered into the editor by pressing the Return key.

Try typing the line below followed by Return:

```
print "Hello"
```

As you type the line, each successive character is printed directly underneath the text cursor, and this cursor is moved one step to the right. You can now edit this line by moving the cursor back to the PRINT statement with the Up arrow key. If you press Return at this point, the line will be re-executed. Notice how the left and right arrow keys move the cursor back and forth along the line. Use these keys to place the cursor over the H, and type:

```
HELP!
```

When you press Return this message will be printed on the screen. The current line can be edited on a character by character basis using the Backspace and Delete keys. In addition, you can delete the entire line with Shift+Delete and join two lines together with Control+J.

The STOS Basic editor provides you with two editing modes: Insert mode and Replace mode. Replace mode is used as the default. In this mode, anything you enter from the keyboard will completely replace the existing text on the screen.

Insert mode is rather different. Instead of overwriting the text, a space for the new character is automatically inserted into the line at the current cursor position. Insert mode is indicated by a thicker cursor and can be toggled on or off using the Insert key. Note that the Replace mode is re-entered whenever the system is reset by the RUN command. Now for an example showing you how this works in practice. Type in the following lines of code.

```
new
10 print "This is a Simple Program"
20 input "What is your name ?";N$
30 print "Hello ";N$
```

This program can be edited using the arrow keys. Incidently you can also place the cursor at the current mouse position by clicking on the left mouse button.

As an example, try changing line 20 to:

```
20 input "What is your Christian name";N$
```

Don't forget to press the Return key after you've edited the line, otherwise it will remain unchanged.

To run your new program type in RUN

The function keys

The upper window contains a brief list of the current function key assignments. Whenever you press one of these keys, the string associated with it will be entered on the screen, just as if you had typed it in yourself. You can also assign a separate set of strings to the shifted versions of these keys, which can be displayed by pressing Shift.

Try entering the following lines:

f2	List
f7	Prints out the current directory
f4	Loads a file from the disc
Shift+f7	Loads all the accessories stored on the current disc

If you play around with these function keys, you may find that the string linked to key number 1 is continually changing. This is because the f1 key is used to hold a copy of your last editor command.

Example:

```
print "Hello"
f1
f1
f1
```

If all this wasn't enough, you can change the function key assigments at any time with the KEY function (See Chapter 10 for more details).

Example:

```
key(3)="boom'"
f3
```

Note that the ' character is used to denote Return.

A list of the current function key assignments is available using the KEYLIST instruction:

KEYLIST *(List the current function key assignments)*

KEYLIST prints out a full list of the strings associated with each of the function keys. The shifted versions of these keys are given numbers from 11-20. Stop listing using either the spacebar, Esc, or Control+C.

f1: KEY LIST'	Last line entered into the system.
f2: list	Lists all or part of a program.
f3: listbank'	Lists banks used by the program.
f4: fload"*.bas"'	Load a Basic program with the file selector.
f5: fsave"*.bas"'	Saves a file using the file selector.
f6: run'	Runs the Basic program.
f7: dir'	Prints out directory of the current disc

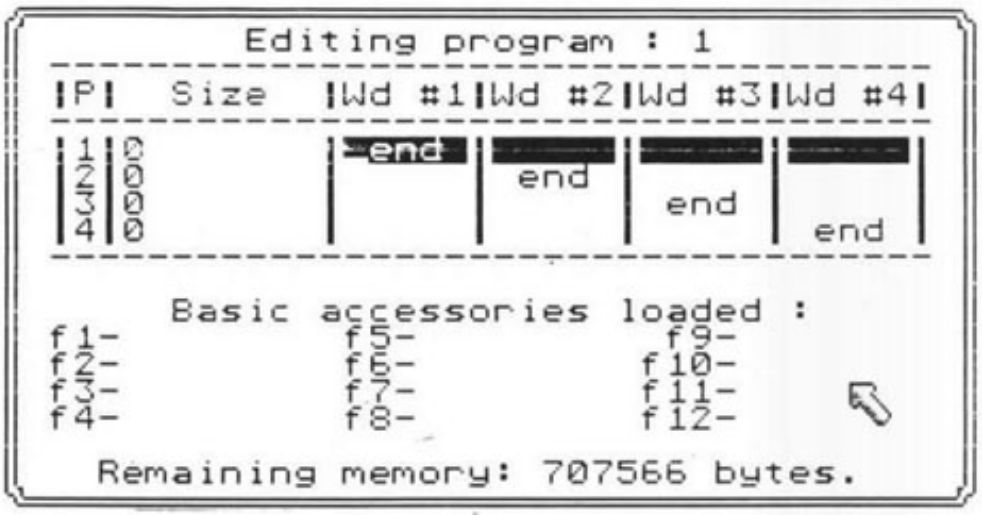
f8: dir$=dir$+\"	Selects a subdirectory. See Chapter 10.
f9: previous	Selects next outer directory.
f10:off'	Turns off sprites.
f11:full'	Sets the editor window to the full screen.
f12:multi 2'	Installs two editor windows.
f13:multi 3'	Installs three editor windows.
f14:multi 4'	Installs four editor windows.
f15:mode 0'	Enter low resolution mode.
f16:mode 1'	Enter medium resolution mode.
f17:accnew:accload "*"	Deletes the current accessories and loads a new set off the disc.
f18:default'	Re-initialise editor screen.
f19:env'	Change colours used by editor.
f20:key list'	List function keys.

The Control keys

The Control keys are a set of commands to the STOS Basic editor which are executed directly from the ST's keyboard. Here is a list of the various control keys and their effects.

Help

This displays the complex looking dialogue box as seen below. There are three distinct parts of this box.



The top section contains a list of the programs currently stored in the ST's memory. STOS Basic allows you to hold up to four Basic programs in memory simultaneously.

The current program is highlighed using a horizontal bar. This bar can be moved up or down with the arrow keys. As you move this bar, the top line changes to indicate the program number which is to be edited. See the section on multiple programs for more information.

The second part of the Help menu displays a list of the accessories installed in the system. These accessories can be executed directly from the help menu by pressing one of the function keys. A list of these accessories, along with their uses can be found on page **55**.

The last line of the help menu displays the amount of memory remaining for the storage of STOS Basic programs. Normally this will be several hundred kilobytes on a standard 520 ST, but if you have loaded all the accessories from the discs, it may well be considerably less.

Control+C

When these two keys are pressed at the same time, any STOS Basic program you are running will be immediately terminated and the control will return back to the editor.

Undo

Pressing this key twice redraws the screen and reinitialises the editor. It is normally used to enable you to edit a program which has corrupted the editor screen, or used to view a line from which an error has occured and forced the program to stop.

Clr

Clears the editor window. Same as CLW.

Up Arrow

Moves the cursor up one line.

Down Arrow

Moves the cursor down one line.

Left Arrow

Moves the cursor one character to the left.

Right Arrow

Moves the cursor one character to the right.

Return

Enters a line at the current cursor position. Exactly the same effect can be achieved by double clicking the left mouse button.

Delete

Deletes the character underneath the cursor.

Shift+Delete

Deletes the line under the cursor.

Backspace

Deletes the character to the left of the cursor, and then moves the cursor one space to the left.

Home

Moves the cursor to the top left hand corner of the screen.

Esc

Enter multi-mode display. See section on multiple programs for more information.

Spacebar

Suspends a listing. Press spacebar again to resume.

Customising the editor

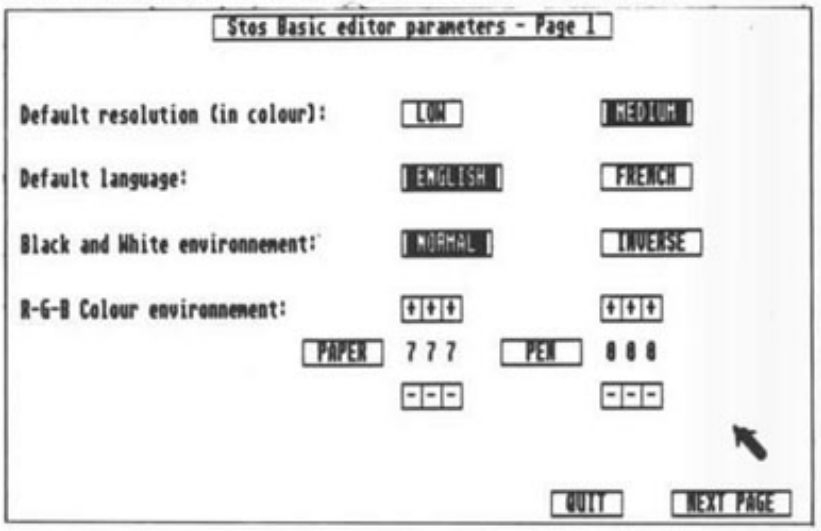
As a default, STOS Basic outputs white text on a black background. You can, however, use any combination of colours you like for the text and background. The easiest way of changing these colours is with the ENV instruction which pages you through 14 different colour schemes. This command is assigned to the shifted f9 key.(Shift+f9)

These colours are retained when you reset the editor using Undo or Default. One major snag with this approach, is that these settings are lost every time you exit from the STOS Basic system. Furthermore, although 14 different options may sound quite a lot, it's really rather restrictive when you realise that both the text and the background can be chosen from a palette of 512 colours. This gives you over 260,000 possible combinations.

Fortunately, the STOS Basic package comes complete with a special configuration program which enables you to customise the system to your own individual requirements. This program can be found on the STOS basic language disc and is called "CONFIG.BAS". It can be loaded and executed by the line:

```
run "CONFIG.BAS"
```

On loading, CONFIG presents you with the following screen:

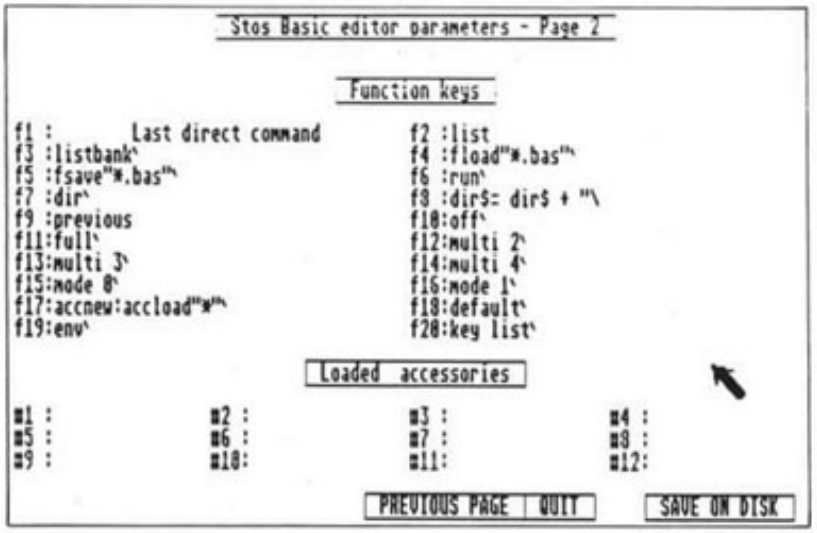


You can select any one of the various alternatives by simply moving the mouse over the appropriate item, and clicking on the left mouse button. If, for example, you wished STOS Basic to enter medium resolution instead of low resolution on loading, you would place the pointer over the MEDIUM option and press the left mouse key. This button would now be highlighted and the LOW option deselected.

You can also use this dialogue to select the colours of the text (PEN), and the background (PAPER). These are specified using a standard RGB format. Each digit in the box corresponds to the strength of either the red, green, or blue components of the colour. These components can take intensities ranging from 0-7. An intensity of zero indicates that none of this component is to be used in the final colour, and a value of 7 denotes the maximum intensity. These numbers can be changed by clicking on the + or - boxes.

Supposing you wanted to set the text colour to yellow, and the background colour to red. In this case, you would set the paper colour to a value of 700, and the pen to 770. (yellow=red+green).

After you have finished with these colour settings, you now need to save them to the disc. Before you can do this, you must first enter the second menu by clicking on the Next Page option. This displays the following dialogue box.



The secondary menu allows you define the default function key assignments, and choose a set of accessories which will be loaded automatically along with STOS Basic. As you move the mouse pointer around on the screen, any function key definitions you pass over are highlighted. These keys can be changed by simply clicking on the left mouse button, and then typing in the new definition.

One interesting possibility is to set the function keys to a list of the 20 most commonly used Basic instructions. This would enable you to type in even the longest STOS Basic programs extremely quickly.

You can also change the accessory list in exactly the same manner. In this case you should enter in the name of the file containing each accessory you wish to be loaded.

Finally these assignments can be saved to the disc by clicking on the Save on Disk option. They will now be automatically set every time you load STOS Basic.

Loading/Saving Basic programs

There are two possible ways you can load a Basic program into STOS Basic. Firstly you can use the normal LOAD option like so:

```
load "CONFIG.BAS"
```

(For a fuller explanation of this command see SAVING and LOADING)

This command works fine if you know the name of the program you wish to load, but often this is not the case. In these circumstances you can use the FLOAD instruction to choose a file using a special file selector.

FLOAD *(Load a file using the file selector)*

FLOAD path$

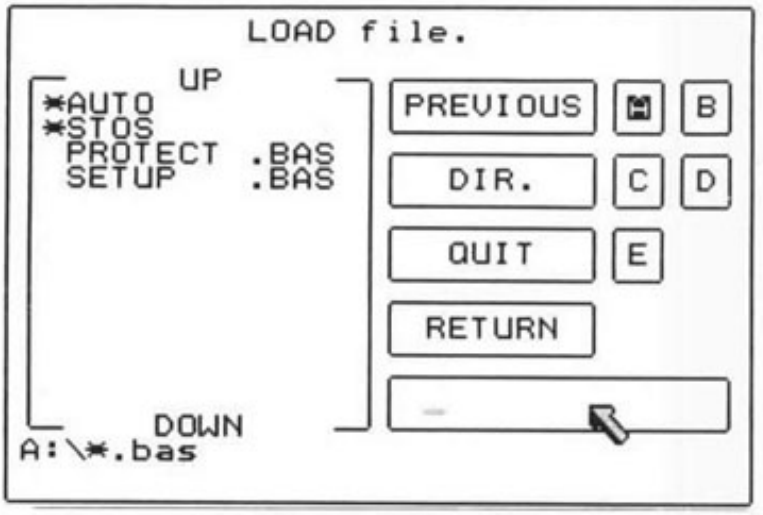
path$ is a string containing the search path. (See DIR)

Example:

```
fload "*.bas"
```

Choose a Basic file to load. Assigned to f4

When you type the above line, a dialogue box will be displayed on the screen. If you are already familiar with the GEM file selector, this should prove fairly self explanatory. If not, then the following diagram should make things a little clearer.



As with the equivalent Gem system, you can choose a file by either clicking on one of the filenames, or typing the name of a file directly into the choice box. This file can then be loaded by either double clicking on the file itself with the left mouse button, clicking on the *Return* box, or pressing Return.

The most obvious difference between this file selector and the Gem version, is the lack of a scroll bar. Instead, you can page through the directory listing by simply clicking on the Up and Down buttons. Also, you can now get a directory listing of the current disc at any time, by clicking on the *Dir* button. This allows you update the directory after you've changed discs.

Note that the * at the front of an item is equivalent to Gem's symbol in that it denotes the existance of a folder. You can enter this folder by clicking on the name. In order to exit back to the outer directory, click on the *Previous* button.

As an example, try loading the CONFIG.BAS file using this file selector.

FSAVE *(Save a Basic file chosen with the file selector)*

FSAVE path$

FSAVE allows you to save a program chosen from a file selector box. As before, *path$* denotes the type of program you wish to save.

Type in the following small program:

```
new
10 print "Executing Line 10"
20 print "Executing Line 20"
30 print "Executing Line 30"
```

Now enter the line:

```
fsave "*.bas"
```

or press function key f5

You will now be presented with the standard file selector. Enter the name of your new file. As you type, the filename is displayed in the current file box. This text can

be edited in the normal way. If you now press Return, your file will be saved to the disc.

You can test this procedure by erasing the program from memory with.

```
new
```

You should now hit f4 to execute the FLOAD command, and double click on the file with your new name. This will then be loaded.

Running a program

RUN *(Execute the current STOS Basic program)*

The standard method of executing a STOS Basic program is using the RUN command. There are three versions of this instruction.

RUN	Run the program starting from the first line.
RUN no	Run the program starting from line number *no*
RUN file$	Load and run the Basic program stored in *file$*

Examples:

Assuming you saved the example file from FSAVE under the filename TEST.BAS, load the file with:

```
load "TEST.BAS"
run
Executing Line 10
Executing Line 20
Executing Line 30

Ok

run 20
Executing Line 20
Executing Line 30

new
run "TEST.BAS"
Executing Line 10
Executing Line 20
Executing Line 30
```

Incidentally, you can also use the RUN command from inside a program. This allows you to chain a number of programs together.

Example:

```
new
10 print "Executing Test"
20 run "TEST.BAS"
30 print "This line is never executed"
```

Any program executed in this way can be terminated using Control+C. You can restart such a program with the CONT command.

CONT *(Restart a program exited by STOP or Control+C)*

CONT re-enters an interrupted program starting from the next instruction. In order for the program to be continued, it must **not** have been changed in the interval between executing the STOP and the CONT.

Example:

```
new
10 for i=1 to 100000
20 print i;
30 next i

run
Control+C      Interrupt the program after a few seconds.
cont           Restart program in the middle of the FOR...NEXT loop.
```

Entering a STOS Basic program

STOS Basic supports two different types of instructions, direct and interpreted. A direct instruction is a command to the editor to perform an action such as listing or saving a program. Most of these direct commands cannot however, be used within a Basic program. Only interpreted instructions such as IF or GOSUB are allowed.

STOS Basic distinguishes between the two sets of operations by checking the first few characters of the current line. If these characters form a line number then you are in interpreted mode, and any direct instructions will cause an error. Otherwise you are in direct mode. Of course, some instructions such as RUN and LOAD can be used in either mode.

In this section, we will be covering the direct mode instructions which allow you to create and modify your STOS Basic programs.

AUTO *(Automatic line numbering)*

The AUTO command is a direct instruction which automatically prints out a new line number every time you press Return. This enables you to enter long Basic programs, without having to continually type in the line numbers. As a default, AUTO starts off at line 10 and increments the line in units of 10.

Look at the example below:

```
auto
10 print "Test of AUTO"
20 goto 10
30 <Return>
run
```

In order to distinguish between the text generated by the computer, and the text entered directly from the keyboard, we've underlined any text which has been typed in by the user. Note how the Return in line 30 was used to exit from this AUTO statement.

Now type the lines:

```
auto
30 print "This line in never reached"
40 <Return>
```

As you can see, the AUTO command automatically started again from line 30. This enables you to jump back into direct mode whenever you wish, and then resume at the point you left off.

It is important to realise that AUTO places you in interpret mode. This means that any direct mode instructions you try to use will cause an error. These instructions include all the normal screen editing operations. Therefore, if you discover a mistake in a line you have just entered, you must exit back to the editor in order to correct it.

Also note that there are a couple of other possible formats to this instruction:

AUTO start	Starts automatic line numbering from line number *start*.
AUTO start,inc	Starts from line *start* and increments each successive line by the number *inc*.

Examples:

```
auto 50
50 print "Test of AUTO"
60

auto 10,1
10 rem First line
11 rem Second line
12
```

RENUM *(Renumber all or part of a program)*

When you're writing a large program, you often end up having to insert many extra lines at various points in your routine. Inevitably, this tends to make your program increasingly messy and hard to read. The RENUM command tidies things up for you by neatly renumbering any or all the lines of your program. The destinations of any GOSUBs or GOTO instructions in the program are automatically amended to take these new line numbers into account.

There are four different ways of using this RENUM command:

RENUM	Starts by setting the first line in your program to 10, and then renumbers each succeeding line in units of 10.
RENUM number	Sets the first program line to *number*, and renumbers all the other lines in increments of 10.
RENUM number,inc	Starts at line *number* and increments each successive line by *inc*.
RENUM number, inc, start-end	Renumbers lines from *start* to *end*, beginning with line number, and incrementing each proceeding line by *inc*.

Note that STOS Basic will not allow RENUM to overwrite any existing parts of the current program.

Example:

```
new
10 print "Example of renumber"
20 goto 50
30 gosub 70
40 stop
50 print " Destination of goto"
60 goto 30
70 print " Destination of gosub"
80 return

renum
list
```

LIST *(List the lines of a Basic program to the screen)*

The LIST command is used to list part or all of the current program to the ST's screen. The format of the instruction is:

LIST	Lists the entire program.
LIST first-	Lists all the lines in the program starting from the line *first*.
LIST -last	Lists the lines from the start of the program to line *last*.
LIST first-last	Lists lines from *first* to *last*.

Note that you can temporarily halt the listing at any time by pressing the spacebar. You can also stop the listing completely using either Esc or Control+C. At the end of the listing, a list of the banks used by the Basic program is appended. The most common use of the list command, is to list a section of the program on the screen for subsequent editing. See LLIST

SEARCH *(Searches for a string in a Basic program)*

SEARCH s$

SEARCH has to be one of the most useful of all the direct instructions, because it allows you to find the position of a string contained within a Basic program. This search string can include any STOS Basic instructions.

Example:

```
load "CONFIG.BAS"
search "print"
3100 paper 1:pen 0:windopen 1,20,6,40,6,10:curs off:print:centre "Please
insert a disc including":print:centre"the stos folder.":print
```

In order to find the next occurrence of the string, you simply type the SEARCH command on its own:

```
search
```

You can also restrict your search to a specific part of the program by adding an optional starting and ending point to the instruction:

SEARCH a$,start-end

start is the line at which the search should begin, and *end* is the line at which it should finish.

The reason why this command is so useful is that you can use it to search through any of the example programs supplied on the STOS Basic disc. Supposing, for instance, you wanted to see how the sprite editor animated its sprites. All you need to do, is type the following lines:

```
load "SPRITES.ACB"
search "anim"
7050 M=0 : gosub 10700 : anim off : sprite off : update : gosub 7325 : loke
start(1)+4,$12 : erase 8 : update off
```

You can repeat this process to find out the precise locations of all the anim instructions in the program by just typing

```
search
```

Another trick is to start any important sections of your program with a line like:

```
999 rem Define sprite
```

This allows you to find the exact position of your routine at any time without having to list through the entire program.

CHANGE *(Change all occurrences of a string in a program)*

CHANGE a$ TO b$ [,start-end]

The CHANGE command searches through a program and replaces any occurrences of the first string with the second. The optional *start* and *end* points define the section of the program which should be changed.

Example:

```
10 AX15B=1
20 for I=1 to 10
30 AX15B=AX15B+AX15B
40 print "The value of variable AX15B is ";AX15B
50 next i
```

Since we've used a rather horrible variable name in this program, we can now change all occurrences of *AX*15B into *COUNT* using the line:

```
change "AX15B" to "COUNT"
```

Listing the program now gives:

```
10 COUNT=1
20 for I=1 to 10
30 COUNT=COUNT+COUNT
40 print "The value of variable COUNT is ";COUNT
```

```
50 next I
```

See also SEARCH.

DELETE *(Delete some or all lines of a program)*

DELETE first-last

The DELETE command is used to selectively erase sections of your Basic programs. If lines *first* and *last* do not exist then this delete operation is not performed.

Example:

```
new
10 rem Line 10
20 rem Line 20
30 rem Line 30
40 rem Line 40

delete 20-30
list
10 rem Line 10
40 rem Line 40
```

Typing a line like:

```
delete 11-31
```

has no effect.

MERGE *(Merge a file into the current program)*

MERGE file$

The MERGE command combines a program stored in the file *file$* with the current program. Existing lines will be overwritten by any new lines with the same number. This instruction is often used to merge a set of subroutines into one complete program.

Debugging a program

Many Basics include a special TRACE command which enables you to step through a program one instruction at a time. The STOS Basic version of this instruction is rather more powerful as it also allows you to track the contents of a list of variables.

FOLLOW *(Track through a STOS Basic program)*

There are five possible formats for the FOLLOW command.

FOLLOW

If the FOLLOW statement is used on its own, the program will halt after every instruction and list the number of the current line. The next line in the program can be stepped through by pressing any key.

FOLLOW first-last	This version of the instruction only follows the program when the lines between *first* and *last* are being executed.
FOLLOW variable list	This takes a list of variables separated by commas and prints them out after every instruction has executed. As before, you can step through the program by pressing any key.
FOLLOW variable list, first-last	Identical to the instruction above, but the variables are only followed when the lines between *first* and *last* are being interpreted.
FOLLOW OFF	Turns off the action of the FOLLOW command.

The FOLLOW instruction has a minimal effect on the current screen, and does not change the position of the text cursor.

Examples:

```
new
10 for X=0 to 10
20 for Y=0 to 10
30 next Y
40 next X
follow X,Y
run
```

Page through the program by pressing any key. To abort the program simply press Control+C

Multiple programs

STOS Basic allows you to have up to four programs in memory at any one time. These may be completely independent of each other. If you suddenly decided to change the configuration of the editor for instance, you could easily load the CONFIG.BAS program into a separate segment of the ST's memory without interfering with your current program.

Example:

```
new
10 print "This is program number ONE"
run
This is program number ONE
```

If you now press the Help key you are presented with a complex looking menu. The top line of this menu has the text *Editing program : 1.* Also, one of the menu lines is inverted. This line indicates the current program segment and is highlighted by the program cursor . Try pressing the Up and Down arrow keys. As the program cursor moves up and down, the program number changes between 1 and 4. Move the program cursor to the second line. The title should now read *Editing program : 2.* You can enter this program segment by pressing the Help key.

Now type:

```
list
```

As you can see, the second program space is empty.

Type the following program:

```
10 print "This is now the second program"
run

This is now the second program.
```

You can now re-enter the first program again using the Help menu. First press the Help key, and then press the Up arrow key once. The title line will now indicate that you are editing program number 1. Exit to this program by pressing Help, and type:

```
run

This is program number ONE
```

So far, we've only used two programs in memory. You can however readily access any of the four programs in exactly the same manner.

MULTI *(Display a number of programs simultaneously.)*

MULTI n

The MULTI command simplifies the process of using multiple programs by dividing the editor window into separate segments, one per program. These programs can be entered with the Help key as before.

Example:

MULTI 2 — Splits the editor window in two.
Top section = Window 1 = Program 1
Bottom section = Window 2 = Program 2
This instruction is assigned to Shift+f2

MULTI 3 — Splits the editor into three sections.
Top section = Window 1 = Program 1
Bottom left section = Window 2 = Program 3
Bottom right section = Window 3 = Program 4
MULTI 3 is assigned to Shift+f3

MULTI 4 — Divides the editor into four quarters. Each window has its own program. Also assigned to Shift+f4

Note that *n* can only take values between 2-4.

As a further example, select segment number 1 with Help and type in:

```
load "CONFIG.BAS"
list
```

Now type:

```
multi 2
```

which splits the window into two and redraws the listing.

You can continue this experiment by typing in the lines:

```
multi 3
```

and

```
multi 4
```

Now type in the command:

```
full
```

which expands the current window to the full screen.

FULL *(Expand current window into the full screen area)*

In expanding the current edit window. *Full* does **not** effect the status of any of the other programs.

Splitting programs in the Editor

You can also use the MULTI command to split a single program into a number of separate sections. This can be done using the Help menu. Position the program cursor over program 1 and press the left and right arrow keys. As you can see, the text cursor is moved between four different boxes on the program line. Move the cursor to the first box and type in 1000 followed by Return. This sets the end point of the first part of the program to line 1000.

If you now exit back to the editor and type MULTI 2, the program will be split into two windows. You can choose between these two windows using the mouse pointer. To see how this works, position the mouse in the top window and click on the left mouse button. The cursor in this window will immediately start flashing, and the window will be activated.

Enter the following line:

```
list
```

This lists all the lines of the program until the line 1000. If you repeat this process in the second window, you will generate a listing of the lines 1000 onwards.

Each box on the program line represents a different section of the listing. You can therefore use this technique to split a program into four separate parts. It is important to note that this has no effect on any existing segments, and you can page through each of the programs stored in memory using the Help menu as usual. All four of these programs can be split in exactly the same way without interferring with each other.

GRAB *(Copy all or part of a program segment into the current program)*

The GRAB command allows you to combine a number of subroutines stored in separate program segments into one complete program. This enables you to test each subroutine in your program independently. The syntax of the GRAB instruction is:

GRAB n — Copy program number *n* into the current program, where *n* ranges from 1 to 4. Any attempt to use the

number of the current program in this instruction will naturally generate an error message.

GRAB n, first-last — Only copies the lines between *first* and *last* into the current program.

See MERGE.

System commands

SYSTEM *(Exit back to Gem)*

The SYSTEM instruction is used to quit from STOS Basic. Note that any programs loaded in STOS Basic which have not been saved to disc will be LOST! You should therefore think carefully before confirming this option with Y.

RESET *(Reset the editor)*

RESET simply reinitialises the editor and redraws the current screen.

DEFAULT *(Reset the editor and redraw current windows)*

DEFAULT redraws any currently defined windows on the screen, and resets the STOS Basic editor. Unlike RESET, DEFAULT can be used either in direct or interpreted mode. This allows it to be utilised at the end of a Basic program to jump back to the editor. The effect of this instruction can also be achieved from the editor by pressing the Undo key twice. Do not confuse this with the DEFAULT function.

NEW *(Erase the current program)*

This command deletes the current program from the ST's memory. It has no effect on any other programs stored in different program segments.

See UNNEW.

UNNEW *(Recover from a NEW and restore the current program)*

UNNEW attempts to recover from the effects of a NEW command, and restore your current program back from the dead. It will only work providing you have not entered any further Basic program lines since the original NEW.

Example:

```
10 rem This line is dead
new
list
unnew
list
```

CLEAR *(Clear all the program variables)*

The CLEAR instruction erases all the variables and all the memory banks defined by the current program. It also repositions the READ pointer to the first DATA statement in the program.

FREE *(Return the amount of free memory)*

FREE returns the number of bytes of memory which is currently available for use by your Basic program. In addition it reorganises the memory space used to hold your string variables. The technical term for this process is garbage collection. Unfortunately, the time taken by this procedure varies exponentially with the number of strings you have defined. This may range from mere milliseconds for small numbers of strings, to several minutes for large string arrays with several thousand elements.

It is important to note that this garbage collection will also occur automatically while your program is running. This is potentially a fairly serious problem as it could lead to your program unexpectedly halting for several minutes. The solution is to call FREE and force this reorganisation when it will cause the least amount of harm.

Example:

```
print free
707536
100 print "Thinking":x=free
```

Note that FREE is equivalent to the FRE(0) function found in many other Basics.

ENGLISH/FRANCAIS *(Choose the language to be used)*

Since STOS Basic originates from France, all system messages are provided in both French and English.

FRANCAIS	Uses French for all subsequent dialogue.
ENGLISH	Uses English for any messages (Default)

FREQUENCY *(Change scan rate from 50 to 60 Hertz)*

This function is only useful is you have a medium resolution monitor capable of scan rates higher than the normal 50 frames per second. If you have a multi-sync monitor, you can use FREQUENCY to improve the quality of the screen display considerably. Note that FREQUENCY also changes the frequency of any interrupts used by STOS Basic to 60 times a second. DO NOT USE THIS FUNCTION WITH A NORMAL TV SET.

UPPER *(Change listing mode to uppercase)*

Normally, any instructions you type into a STOS Basic program are listed in lower case, and any variables in upper case. The UPPER directive reverses this format.

Example:

```
new
10 n=10
20 PRINT "The Value of N is ",n

list
10 N=10
20 print "The Value of N is ",N
```

```
upper
list
10 n=10
20 PRINT "The Value of N is ",n
```

LOWER *(Change Editor mode to lower case)*

LOWER returns the listing format back to the default case. Any variables will now be listed to the screen or printer in upper case, and instructions will be output in lower case.

Naming conventions for variables

The names of STOS Basic variables need to conform with a number of rules. Firstly, each variable name must begin with a letter. Also, the names must not contain any of the following Basic keywords.

TO, STEP, THEN, ELSE, XOR, OR, AND, GOTO, GOSUB, MOD, AS

All other keywords such as RUN or POKE are, however, perfectly legal.

Examples of legal variable names:

A, RUNE$, IPOKE, TEST, ZZ99, C5#

Here are a few examples of illegal names. We've underlined the illegal bits to make things clearer.

CAST, 5C, SORT, BAND$, MODERN#, TOAD

The maximum length of these variable names is 31 characters. Note that the # and $ suffixes denote the type of variable.

Types of variables

STOS Basic allows you to use three different types of variables in your programs.

Integers

Unlike most other Basics, integers are used by default. Since integer arithmetic is generally much faster than the more normal floating point operations, this strategy can often improve the speed of Basic programs considerably. Each integer is stored in four bytes, and can range from:

-2147483648 to + 2147483648

Examples of integer variables:

A, NUMBER, HELLO

Real numbers

These are suffixed with a # character. They correspond directly to the double precision floating point variables used in other versions of Basic. Each real variable is stored in eight bytes, and can range between:

-1.797692 E+308 and +1.797693 E+307

These real numbers are accurate to a precision of 16 decimal digits.

Examples of floating point variables:

P#, NUMBER#, TEST#

String variables

String variables are always suffixed with the $ character, and can range from 0-65500 characters long. They are not terminated with a chr$(0).

Examples of string variables:

NAME$, TEST$, TEL$

Arrays

Any of the above variable types can be incorporated into a table known as an array These arrays can be created using the DIM instruction.

DIM *(Dimension an array)*

DIM is used to set up a table of variables. These tables may consist of any number of dimensions you like, but each dimension is limited to a maximum of 65535 elements.

Example:

```
10 dim A$(10),B(10,10),C#(10,10,10)
```

In order to access an individual element in this array, you simply type the array name followed by the index number enclosed between round brackets (). The following small example should make this a little clearer:

```
new
10 dim NAME$(10),AGE(10)
20 for I=0 to 10
30 input "What is your Name";NAME$(I)
40 input "What is your Age";AGE(I)
50 next I
60 print "NAME AGE"
70 print "======================="
80 for I=0 to 10
90 print NAME$(I),AGE(I)
100 next I
```

It is important to note that the element numbers of these arrays always start from zero.

See MATCH and SORT.

Constants

As a default, all numeric constants are treated as integers. Any floating point

assignments to an integer variable are automatically converted to a whole number before use.

Examples:

```
A=3.1411:print A
3

print 19/2
9
```

In addition to the usual decimal notation, you can also use either binary or hexadecimal expressions.

Binary numbers are signified by preceeding them with a % character, and hexadecimal numbers are denoted by a $ sign. Here are a few examples of the various different ways the number 255 could be expressed.

Decimal:	255
Hexadecimal:	$FF
Binary:	%11111111

Note that any numbers you type into STOS Basic are converted into a special internal format. When you list your program, these numbers are expanded back into their original form. Since STOS Basic prints all numbers in a standard way, this will often lead to minor discrepances between the number you entered, and the number which is displayed in the listing. The VALUE of the number will however, remain completely unchanged.

Floating point constants are distinguished from integers by a decimal point. If this point is not used, then the number will always be assumed to be an integer, even if this number occurs inside a floating point expression. Take the following example:

```
new
10 for i=1 to 10000
20 A#=A#+1
30 next i
```

In this program, the "1" in line 20 is stored as an integer. Since the conversion between integer and floating point numbers takes place each time the line executes, this program will be inherently slower than the equivalent routine below.

```
new
10 for i=1 to 10000
20 A#=A#+1.0
30 next I
```

This program executes over 25% faster than the original one because the constant in line 20 is now stored in floating point format. You should therefore always remember to place a decimal point after a floating point constant even if it is a whole number.

Incidentally, if you mix floating point numbers and integers in an expression, the result will always be returned as a floating point number.

Example:

```
print 19.0/2
9.5
```

```
print 3.141+10
13.141
```

Arithmetic operations

The following arithmetic operations can be used in a numeric expression.

^	Power
/ and *	Divide and multiply
MOD	Modulo operator (Produces remainder of a division)
+ and -	Plus and minus
AND	Logical AND
OR	Logical OR
XOR	Logical XOR

We've listed these operations in ascending order of their priority. This priority refers to the sequence in which the various sections of an arithmetic expression are evaluated. Operations with the highest priority are always calculated first. Here is an example of how this works in practice.

```
print 10+2*5-8/4+5^2
```

This evaluates in the following order:

```
5^2      = 5*5 = 25
2*5      = 10
8/4      = 2
10+10    = 20
20-2     = 18
18+25    = 43
```

If you wanted this to evaluate differently, you would simply enclose the parts of the expression you wished to execute first in round brackets:

```
print (10+2)*(5-8/4+5)^2
```

This gives the result 12*(8^2) or 12*64 or 768. As you can see, the addition of just two pairs of brackets has changed the sense of the expression entirely.

While on the subject of arithmetical operations, it's worth mentioning two useful functions: INC and DEC.

INC *(Add 1 to an integer variable)*

INC var

INC adds one to an integer variable using a single 68000 instruction. It is logically equivalent to the expression *var=var+1*, but is much faster.

Example:

```
new
10 timer=0
20 print "Increment A with A=A+1"
30 for I=1 to 10000
40 A=A+1
```

```
50 next I
60 print "Took ";timer/50.0;" Seconds"
70 timer=0
80 print "Increment A with INC instruction"
90 for I=1 to 10000
100 inc A
110 next I
120 print "Took ";timer/50.0;" Seconds";

run
```

It should be apparent that the second version of the FOR...NEXT loop executes considerably faster.

DEC *(Subtract 1 from an integer variable)*

DEC var

This instruction subtracts one from the integer variable *var.*

Example:

```
A=2
dec A
print A
1
```

String operations

Most modern Basics allow you to add two strings together like this:

```
A$="STOS"+" Basic"
print A$
STOS BASIC
```

In addition STOS Basic also lets you perform subtraction with string variables as well. This operation works by removing all occurrences of the second string from the first.

Examples:

```
print "STOS BASIC"-"S"
TO BAIC
print "STOS BASIC"-"STOS"
BASIC

print " A String of Char acters"-" "
AStringofCharacters
```

Comparisons between two strings are performed on a character by character basis using the Ascii codes of the characters.

Examples:

```
"AA" < "BB"
"Filename"="Filename"
"X&" > "X#"
"HELLO" < "hello"
```

Common string functions

LEFT$ *(Return the leftmost characters of a string)*

LEFT$(v$,n)

There are two distinct forms of this command. The first version of LEFT$ is configured as a function and returns the first *n* characters in the string expression *v$*.

Examples:

```
print left$("STOS Basic",4)
STOS
a$=left$("0123456789ABCDEF",10)
print A$
0123456789

10 input "Input a string";V$
20 input "Number of characters";N
30 print left$(V$,N)
40 goto 10
```

There's also a different variant of LEFT$ implemented as an instruction.

LEFT$(v$,n)=t$

This instruction sets the leftmost *n* characters in *v$* to *t$*. If *t$* is longer than *n*, it is truncated to the appropriate length. Note that unlike the LEFT$ function *v$* must be a string variable rather than an expression.

Example:

```
10 A$="** Basic"
20 left$(A$,4)="STOS"
30 print A$
run
STOS Basic
```

RIGHT$ *(Return the rightmost character of a string)*

RIGHT(v$,n)

Return the rightmost character in *v$*. RIGHT$ is a function which reads *n* characters from the string expression *v$* starting from the right.

Examples:

```
print right$("STOS Basic",5)
Basic

A$=right$("0123456789ABCDEF",10)
print A$
6789ABCDEF

new
10 input "Input a string";V$
20 input "Number of characters";N
```

```
30 print right$(V$,N)
40 goto 10
```

As with LEFT$ there's also another version of RIGHT$ set up as a Basic instruction.

RIGHT$(v$,n)=t$ — Set rightmost n characters of *v$* to *t$*. Note that *v$* should always be a string variable, and that excess characters in *t$* are omitted.

Example:

```
new
10 A$="STOS **"
20 right$(A$,5)="Basic"
30 print A$

run
STOS Basic
```

See LEFT$, MID$

MID$ *(Return a string of characters from within a string expression)*

MID$(v$,s,n) — The MID$ function returns the middle section of the string *v$*. *s* denotes the number of character at the start of this substring, and *n* holds the number of characters to be fetched. If a value of *n* is not specified in the instruction then the characters are read up to the end of the string *v$*.

Examples:

```
print mid$("STOS Basic",6)
Basic
print mid$("STOS Basic",6,3)
Bas

new
10 input "Input a string";V$
20 input "Starting Position, Number of characters";S,N
30 print mid$(V$,S,N)
40 goto 10
```

There's also a MID$ instruction.

MID$(v$,s,n)=t$ — This version of MID$ sets *n* characters in *v$* starting from *s* in the string *t$*. If a value of *n* is not included in this instruction, then the characters are replaced up to the end of *v$*.

Examples:

```
A$="STOS **"
mid$(A$,6)="Magic"
print A$
STOS Magic
```

```
mid$(A$,6,3)="Bas"
print A$
STOS Basic

new
10 input "Input a target string";V$
20 input "Input a substring";T$
30 input "Starting Position, Number of characters";S,N
40 mid$(V$,S,N)=T$
50 print V$
60 goto 10
```

INSTR *(Search for occurences of a string within another string)*

INSTR allows you to search for all occurrences of one string inside another. It is especially useful for adventure games as it enables you to split a line of text into its individual words. There are two forms of the INSTR function.

INSTR(d$,s$) — This searches for the first occurrence of *s$* in *d$*. If the string is found, then the position of this substring is returned by the function, otherwise a value of 0 is returned.

Examples:

```
print instr("STOS Basic","STOS)
6
print instr("STOS Basic","S")
1
print instr("STOS Basic","FAST")
0

new
10 input "String to be searched";D$
20 input "String to be found";S$
30 X=instr(D$,S$)
40 if X=0 then print S$;" not found"
50 if X<>0 then print S$;" found at position ";X
60 goto 10
```

INSTR(d$,s$,p) — This version of INSTR finds the first occurrence of s$ in d$ starting from character number p.

Examples:

```
print instr(STOS BASIC","S",2)
4
```

You can change the above example to this new form of INSTR by typing the lines:

```
25 input "Starting position";P
30 X=instr(D$,S$,P)
```

Here is an example which splits a line of text separated by spaces, into its component words.

```
10 print "Please type a string of characters" : input P$
```

```
20 I=0
30 repeat
40 P1=instr(P$," ",P)
50 if P1<>0 then L=P1-P else L=len(P$)-P+1
60 print "Word number ",I;" = ";mid$(P$,P,L) : P=P1+1 : inc I
70 until P1=0
```

Array Operations

SORT *(Sorts all elements in an array)*

SORT a$(0)

The SORT instruction allows you to sort all the elements in an array into ascending order amazingly quickly. This array can be composed of either strings, integers, or floating point numbers. The *a$*(0) indicates the starting point of the table to be sorted. This starting point must always be set to the first item in the array (item zero).

Example:

```
10 dim A(25)
20 P=0
30 repeat
40 input "Input a number (0 to stop)";A(P)
50 inc P
60 until A(P-1)=0 or P>25
70 sort A(0)
80 for I=0 to P-1
90 print A(I)
100 next I
```

SORT is often used in conjunction with the MATCH instruction to perform complex string searches.

MATCH *(Find the closest match to a value in an array)*

MATCH (t(0),s)

The MATCH function searches through a sorted table, and returns the item number in which the value *s* was found. If *s* is not found, then MATCH returns a negative number. The absolute value of this number contains the index of the first item which was greater than s. Providing the array is of only one dimension, it can be of type string, integer or real. Before MATCH can be used the array should always be sorted using the SORT command.

Example:

```
new
10 read N
20 dim D$(N)
30 for I=1 to N
40 read D$(I)
50 next I
60 sort D$(0)
```

```
70 input A$
80 if A$="!" then for I=1 to N : print D$(I) : next I : goto 70
90 POS=match(D$(0),A$)
100 if POS>0 then print "found",D$(POS);" in record ";POS
110 if POS<0 and abs(POS)<=N then print A$,"not found. Closest to
",D$(abs(POS))
120 if POS<0 and abs(POS)>N then print A$,"not found. Closest to";D$(N)
130 goto 70
140 data
10,"adams","asimov","shaw","heinlien","zelazny","foster","niven"
150 data "harrison","pratchet","dickson"
```

Note that the MATCH instruction could be used in conjunction with INSTR to provide a powerful PARSER routine which could form the basis of an Adventure game.

Memory banks

STOS Basic includes a number of powerful facilities for the manipulation of sprites, screens and music. The data required by these functions needs to be stored along with the Basic program. STOS Basic uses a special set of 15 sections of memory for this purpose called Banks. Each Bank is referred to by a unique number ranging from 1-15. Many of these banks can be used for all types of data, but some are dedicated solely to one sort of information such as sprite definitions. Every program stored in the ST's memory has its own separate set of Banks.

There are two different forms of memory bank: Permanent and temporary. Permanent banks only need to be defined once, and are subsequently saved along with your program automatically. Temporary Banks however, are much more volatile and are reinitialised every time a program is run. Furthermore, unlike permanent banks, temporary banks are erased from memory by the CLEAR command.

Types of memory bank

Each memory bank can be one of following different types.

Class	Stores	Restrictions		Type
Sprites	Sprite definitions	Only bank 1	(1)	Permanent
Icons	Icon definitions	Only bank 2	(1)	Permanent
Music	Music	Only bank 3	(1)	Permanent
3D	Future 3D extension	Only bank 4	(4)	Permanent
Set	Holds new character sets	Banks 1-15		Permanent
Screen	Stores a complete screen	Banks 1-15		Temporary
Datascreen	Stores a screen	Banks 1-15		Permanent
Work	Temporary workspace	Banks 1-15		Temporary
Data	Permanent workspace	Banks 1-15		Permanent
Menu	Menu lines	Bank 15	(2)	Temporary
Program	Machine-code program	Banks 1-15	(3)	Varies

Footnotes:

(1) Bank is not really general purpose. It is allocated automatically by the appropriate accessory, or when a bank of this type is loaded.

(2) Reserved automatically by MENU commands. Usable by programs which don't use menus.

(3) Reserved as either Work or Data. Renamed when program loaded into bank. See LOAD.

(4) Reserved for future expansion.

You can get a list of the status of the Banks which are currently being used by a program with the LISTBANK command.

LISTBANK *(List the banks in use)*

LISTBANK lists the numbers of the banks currently reserved by a program, along with their location and size.

Example:

```
load "BULLET.BAS"

listbanks
Reserved memory banks:
1   sprites    S:$055000   E:$066500   L:$011500
3   music      S:$066500   E:$067300   L:$000E00
7   data       S:$067300   E:$069300   L:$002000
8   program    S:$069300   E:$069B00   L:$000800
9   data       S:$069B00   E:$06A200   L:$000700
10  data       S:$06A200   E:$06A900   L:$000700
11  data       S:$06A900   E:$06AF00   L:$000600
12  data       S:$06AF00   E:$06C000   L:$001100
13  data       S:$06C000   E:$06FF00   L:$003F00

S: = The start address of the bank.
E: = The end address of the bank.
L: = The length of the bank.
```

As a default all these values are printed out in hexadecimal notation. You can, however, change the format of the listings into decimal using the command HEXA OFF

HEXA ON/OFF *(Toggle hexadecimal listing)*

HEXA OFF	Sets bank listings to decimal notation.
HEX ON	Sets bank listings to hexadecimal format.

Example:

```
load "BULLET.BAS"
hexa off
listbanks
Reserved memory banks:
1   sprites    S:348160    E:419072    L:7091
3   music      S:419072    E:422656    L:3584
7   data       S:422656    E:430848    L:8192
8   program    S:430848    E:432896    L:2048
9   data       S:432896    E:434688    L:1792
```

```
10 data        S:434688  E:436480  L:1792
11 data        S:436480  E:438016  L:1536
12 data        S:438016  E:442368  L:4352
13 data        S:442368  E:458496  L:16128
```

RESERVE *(Reserve a bank)*

Any banks used by the sprites, music, icons, 3D extensions, and the menus are allocated automatically by the system. The RESERVE command allows you to allocate any other banks which you require. Each different type of bank has its own individual form of the RESERVE instruction.

RESERVE AS SCREEN bank	Reserves a temporary bank of memory for a screen. This bank is always 32k long.
RESERVE AS DATASCREEN bank	Reserves a permanent bank of memory 32k long for use as a screen. This screen is saved along with your program, so it's great for title screens. See Chapter 7 for examples of this instruction in action.
RESERVE AS SET bank,length	Reserves a permanent bank of memory *length* bytes long for use as a character set. See Chapter 8.
RESERVE AS WORK bank,length	Reserves a temporary bank for use as a workspace *length* bytes long.
RESERVE AS DATA bank,length	Reserves a permanent bank of memory *length* bytes long for use as a workspace.

Note that *bank* may be any number between 1-15. Since banks 1 to 4 are normally reserved by the system, it's wisest to leave these banks alone. *Length* is automatically rounded up to the nearest 256 byte page. The only other limit to the length of a bank is the amount of available memory.

Type the following lines:

```
new
hexa off
reserve as screen,5
listbank
Reserved memory banks:
5 screen  S: 950016 E: 982784 L: 32768
```

This reserves bank number 5 as a temporary screen. Now type:

```
clear
listbank
```

As you can see, bank 5 has now been completly erased. In order to create a more permanent bank, enter:

```
reserve as datascreen 5
```

```
listbank
clear
listbank
Reserved memory banks:
5  dscreen S: 950016 E: 982784 L: 32768
```

Bank 5 is totally unaffected by the clear command. We'll now demonstrate how this screen can be loaded with real data.

screen copy logic to 5	**Copies the current screen to bank 5.**
cls	**Erase screen**
screen copy 5 to logic	**Copies bank 5 back to current screen, and restores it.**

For more information about SCREEN COPY see Chapter 7.

Copying banks

When using these memory banks, it's often useful to be able to transfer the contents of one bank to another. This can be done with a special BCOPY command.

BCOPY *(Copy the contents of a bank to another bank)*

BCOPY #source TO #dest

BCOPY copies the entire contents of bank number *source* into bank number *dest*. As usual *source* and *dest* can range from 1-15

Example:

```
BCOPY 5 TO 6 Copies bank 5 into bank 6
```

BGRAB *(Copy some or all banks from a program to the current program)*

BGRAB prgno [,b]

BGRAB copies one or more banks stored at program number *prgno* into the current program. Program numbers between 1-4 denote one of the four programs which can be stored in memory at any one time. Numbers from 5-16 represent an accessory.

If the optional bank number *b* is not included, then all the banks attached to program number *prgno* are copied into the current program, and any other banks of memory which are linked to this program are erased. Otherwise, the bank number specifies one bank which is to be transferred into the current program. All other banks remain unaffected.

This instruction is used to great effect by many of the accessories on the disc.

Deleting banks

ERASE *(Delete a bank)*

ERASE b

ERASE deletes the contents of a memory bank *b*. As usual *b* can range from 1-15. Any memory used by this bank is freed for use by your program.

Bank parameter functions

=START *(Get the start address of a bank)*

bs=START(b)

This function returns the start address of bank number *b* in the ST's memory.

START(b)	Returns the start of bank *b* in the current program
START(prgno,b)	Returns the start of the bank number *b* in program *prgno*.

Note that *b* can range from 1-15, and *prgno* from 1-16. Program numbers greater than 4 refer to accessories.

Example:

```
reserve as screen 10
print start(10)
```

=LENGTH *(Get the length of a bank)*

bl=LENGTH(b)

This function returns the length in bytes of bank number *b*. If a value of zero is returned by LENGTH, then bank *b* does not exist.

LENGTH(b)	Gets the length of bank *b* in the current program.
LENGTH(prgno,b)	Gets the length of bank *b* in program number *prgno*.

Example:

```
new
reserve as screen 5
print length(5)
32768
erase 5
print length(5)
0
```

Saving and loading

SAVE *(Save part or all of a STOS Basic program)*

The SAVE instruction provides a general and straightforward way of saving a STOS Basic program on to the disc. Unlike the equivalent instruction found in most other versions of Basic, STOS also allows you to save a variety of other types of information. This is determined by the extension of the filename used in the SAVE command. Here is a summary of the various data types, along with their extensions.

Type of Information	Extension	Comments
Basic programs	.BAS	Normal Basic program
Accessories	.ACB	Load using ACCLOAD
Images	.PI1, PI2 or PI3	Degas format screen shot.
	.NEO	Neochrome format. Only in low resolution.
Memory banks	.MBK	One memory bank.
	.MBS	All current banks.
Basic variables	.VAR	All currently defined variables
Listings	.ASC	In Ascii format
RUN-ONLY programs	.PRG	Executable directly from desktop.

If none of these extensions are used, then STOS adds .BAS to the Filename automatically, and saves the current Basic program on to the disc. Any existing program of the same name will be renamed with the extension .BAK.

We'll now discuss each of the possible options in a little more detail.

SAVE "Filename.BAS"

This saves the current program on to the disc under the name filename.BAS. If a file with the same name already exists, this is overwritten.

SAVE "Filename.ACB"

Saves the Basic program as an accessory. This program can be loaded using ACCLOAD, and accessed from the HELP menu at any time.

SAVE "Filename.PI1"[,address of screen]
SAVE "Filename.PI2"[,address of screen]
SAVE "Filename.PI3"[,address of screen]

This instruction saves a copy of the screen to the disc in Degas format. The different extensions indicate the resolution of the image.

.PI1 = Low resolution
.PI2 = Medium resolution
.PI3 = High resolution

The Screen address is optional. If it is omitted from the statement, then the current screen will be saved to the disc.

Example:

```
save "screen.PI1"
cls
load "screen.PI1"
```

See LOAD.

Any screen saved in this manner can be subsequently edited directly from Degas.

SAVE "Filename.NEO"

Saves a low resolution screen in Neochrome format. This file can be either loaded into a Basic program, or modified from within Neochrome.

save "Filename.MBK",b

This version of SAVE stores the memory bank with number *b* on to the disc. It can be loaded back again using LOAD. An example of this function can be found in the section on LOAD.

save "Filename.MBS"

Saves all the banks allotted to the current program in one large file. See LOAD ".MBK" for more details.

save "Filename.VAR"

SAVE "Filename.VAR" provides you with the ability to save all the currently defined variables directly on to the disc. Again see LOAD for an example of this function.

save "Filename.ASC"

Lists the Basic program to a file in Ascii format. This file can now be edited outside STOS Basic by a wordprocessor or a text editor. Note that the Banks of memory are not output by this function. We've used this instruction extensively in the creation of this manual. Most of the included listings are derived directly from the original programs.

BSAVE *(Save a block of memory in binary format)*

BSAVE file$, start to end

The memory stored between *start* and *end* is saved to the file *file$*. The data is saved out as it is in memory with no special formatting. You can use this function for various tasks one of which would be to save out a character set from bank 5.

bsave "\STOS\8X8.CR0", start (5) to start (5)+length (5)

See BLOAD

Run-only programs

save "Filename.PRG"

This option saves a version of your program in a special format which allows it to be loaded and executed straight from the Gem desktop. In order to use this function, you should first prepare a disc using the STOSCOPY.ACB accessory. This makes a copy of the entire \STOS\ directory on the disc. This disc can now be used to hold your run-only program. NEVER SAVE A RUN-ONLY PROGRAM ON THE ORIGINAL SYSTEM DISC!

When you save one of these programs, two files with the same name are created on the disc. One file has the extension .BAS and is stored in the \STOS\ folder. The second file lies outside the folder, and has the .PRG extension. It is this file which can be executed from the GEM desktop. When a run-only program terminates or an error occurs, it immediately returns to Gem.

As an example, generate a disc with the correct files using a freshly formatted disc in conjunction with STOSCOPY.ACB accessory. Now load the sprite editor into memory using the line:

```
load "sprites.acb"
```

Place the save disc into the drive, and type:

```
save "sprites.prg"
```

At this point STOS Basic will ask you to confirm that you really wish to save this program. Enter Y or y at this prompt.

You have now installed a run only version of the sprite generator, which can be executed directly from the Gem desktop. To test this, quit from STOS Basic using the SYSTEM command, and double click on the file sprites.prg. This file is now loaded, and the sprite editor is run, just as if you were executing it directly from STOS Basic. This program can be terminated using the menu option QUIT or Control+C.

Notes:

1. Any attempt to execute the STOS Basic editor from a run-only program will crash the ST completely.

2. The files PIC.PI1 and PIC.PI3 in the STOS folder contain low and high resolution pictures which will be displayed automatically during loading. If you like, you can omit these files from the disc to save space.

3. The default colours used by your program will be the standard ones used by the Gem Desktop, and not the normal STOS Basic colours.

4. Any of your own programs installed as RUN ONLY may be freely distributed or sold providing you acknowledge that they were written in STOS Basic and use the protect accessory when giving the disc to anyone who has not bought a copy of STOS Basic.

5. If you place the run-only program in the \AUTO\ folder it will load and run automatically, whenever the disc is booted up.

6. For more information see Appendix B.

LOAD *(Load part or all of a STOS Basic program)*

The LOAD instruction complements SAVE by allowing you to enter either a program or data file from the disc. Here is a list of the various types of files which may be loaded using this command.

Type	Extensions allowed
Basic programs	.BAS, .BAK, .ACB, .ASC
Images	.NEO, .PI1, .PI2, .PI3
Memory banks	.MBK, .MBS
Variables	.VAR
Machine-code programs	.PRG

See SAVE for a fuller discussion of these extensions.

Basic Programs

LOAD "Filename"

Loads a Basic program. Assumes the extension ".BAS"

LOAD "Filename.BAS"

Loads a Basic program with the extension ".BAS". Identical to LOAD "filename"

Example:

```
load "config.bas"
run
```

LOAD "Filename.BAK"

Loads a backup of a Basic program created using the SAVE "Filename" instruction.

LOAD "Filename.ACB"

This loads an accessory as a normal Basic program. It can now be edited and debugged in the usual way.

Example:

```
load "type.acb"
list
```

LOAD "Filename.ASC"

This option lets you load an Ascii version of a Basic program, created using either a text editor, or another version of Basic. Note that this program must have line numbers, and be in plain Ascii. First Word users should turn the WP option off before exporting a program into STOS Basic. It is important to realize that this instruction does not erase the current program. Instead the new file is merged with this program.

The ability to load a Basic program in this format can be used to allow you to generate new STOS Basic listings within a Basic program. This has been used by the sprite editor to dump the contents of a sprite bank onto the disc in the form of a list of DATA statements.

LOAD "Filename.MBK"[,b]

This loads a single data file into a memory bank. If the optional destination of this data is included, then the file is loaded directly into Bank number *b*, where *b* can range from 1-15. Otherwise the file is loaded back into the bank from which it was saved. Note that any existing data in this bank is erased during this loading process. Furthermore, the LOAD instruction automatically reserves a bank of the appropriate type if it has not already been defined.

Examples:

```
new
load "sprdemo.mbk
load "musdemo.mbk
load "icondemo.mbk
listbank
```

LOAD "Filename.MBS"

Loads a series of banks stored in a single file. These banks are loaded directly into their original bank numbers. If these banks already exist, the old versions are erased.

Place a fresh disc into the drive, and type:

```
save "BANKS.MBS"
new
listbank
load "BANKS.MBS"
listbank
```

As you can see, all three banks have been loaded in one operation.

Variables

LOAD "Filename.VAR"

This loads a list of variables stored on the disc using SAVE "filename.VAR". Any currently existing variables are replaced. Note that this instruction affects ALL the variables in the program.

Example:

```
new
10 dim A(100)
20 for X=1 to 100
30 A(X)=X
40 next X
50 save "numbers.VAR"
```

Run this program with a disc in the drive. Now type in:

```
new
load "numbers.VAR"
for X=1 to 100:print A(X):next x
```

See how the array *A* has been automatically defined by the load operation.

Images

```
LOAD "Filename.PI1"[,address of screen]
LOAD "Filename.PI2"[,address of screen]
LOAD "Filename.PI3"[,address of screen]
```

The above commands load a Degas format picture file from the disc. If the address of the screen is not included in the statement, then this image will be loaded into the current screen. Otherwise it will be loaded into the screen at address. Normally this address will point to the start of a memory bank defined as either a SCREEN or DATASCREEN.

Remember that *PI1* denotes a low resolution screen, *PI2* medium resolution, and *PI3* high resolution.

Example:

Place the disc containing the \STOS folder into your disc drive and type in:

```
cls
```

If you have a colour monitor you can now type:

```
mode 0
load "\STOS\PIC.PI1
```

and for a monochrome monitor:

```
load "\STOS\PIC.PI3"
```

These commands load the STOS title screen into the ST's memory.

BLOAD *(Load binary information into a specified address or bank)*

This function load in binary data without altering the incoming information. There are two forms of this function.

BLOAD file$,addr	The file *file$* will be loaded into the address addr.
BLOAD file$, #bank	*file$* is loaded into bank, thus the address from which the data resides once it has been loaded is the start address of *bank*. This start value can be found with the command:

bkaddr = start (bank)

To see an example of this command insert the accessory disc and type in the line:

```
bload "mouse.acb", physic
```

which loads in the mouse accessory at the memory address of the physical screen.

See BSAVE.

Machine-code programs

```
LOAD "Filename.PRG",b
```

This instruction allows you to load a machine-code program into a memory bank number *b*. Any program you wish to use in this manner should be stored in TOS relocatable format, and must be placed in a file ending with the ".PRG" extension. DO NOT TRY TO USE GEM-BASED PROGRAMS FOR THIS PURPOSE! You should also avoid accessing any of the memory management functions from Gemdos. All other functions may be used, providing you take care.

You can call one of these functions using the CALL instruction like so:

```
CALL START (Bank number)
```

See Appendix C for more details.

Note that when you copy a bank containing a program into another bank, this is automatically relocated for you.

The accessories

The STOS Basic accessories are special programs which lie dormant in the ST's memory until you call them up using the Help key.

ACCLOAD *(Load an accessory)*

Before you can use one of these accessories you must first load it into memory using the ACCLOAD command.

```
accload "name"
```

ACCLOAD loads the accessory from the file *name* into memory. Any normal Basic programs you have entered will be completely unaffected.

Example:

```
accload "sprites.ACB"
```

You can use this function to load all the accessories stored on a disc into memory at once. In order to do this, simply specify a name of *

Example:

```
accload "*"
```

Note that you can also use CONFIG.BAS to install a list of these accessories permanently. This is very wasteful of memory and should be used with caution by users restricted to a standard 520ST.

ACCNEW *(Remove all currently installed accessories)*

ACCNEW erases all the accessories from memory. It is often used in conjunction with ACCLOAD to remove any unwanted accessories before loading a new one.

Example:

```
accnew:accload "*"
```

See also ACCNB.

Calling an accessory

A list of the accessories currently available can be found by pressing the Help key at any time. This displays a list of function keys alongside the accessories. In order to call the accessory, simply press the appropriate key. Note that these keys only call up the accessory from the HELP menu.

The sprite definer

This accessory is stored in the file SPRITES.ACB and provides a quick and convenient method of creating or editing lists of sprites. A full explanation of this program can be found in Chapter 4.

The character definer

The character definer in FONTS.ACB is used to create one of 13 user-defined

character sets. These sets can be accessed within a STOS Basic window, or can directly replace the existing character set. See Chapter 8 for more details.

The icon definer

ICONS are special 16x16 characters which can be displayed in maps, or incorporated into menus. The ICON definer in ICONS.ACB allows you to create up to the 255 of these objects.

The music creation utility

MUSIC.ACB holds a powerful and effective tool for composing music or sound effects that can be used within any STOS Basic program. Any music created with this utility can operate independently of the rest of the program. See Chapter 5 for a thorough examination of this accessory.

Compact

The screen compactor is a simple way of compressing a screen into a small space. Typical compaction ratios vary from 30 per cent to up to 75 per cent. The COMPACT.ACB accessory provides an effective method of performing these compressions, and saving the results on to the disc. These files can then be expanded with the UNPACK instruction. See Chapter 7.

Scan

Opens a window in the centre of the screen and prompts you for a keypress. The Scancode and the Ascii code of this key are then displayed.

Ascii

Displays an Ascii table on the screen. Note that the row and column numbers are in hexadecimal. Convert to decimal using $.

Example:

```
print $FF
```

Mouse

As you move the mouse pointer around on the screen, the current X and Y coordinates are displayed in the Mouse window. To exit from this accessory click once on either of the mouse keys.

Type

Prints an Ascii file on the disc to either the screen or the printer.

Stoscopy

This accessory copies the \STOS\ folder along with its contents on to a new disc. Since this function requires you to input the system disc into the current drive, it's a good idea to set the write protect tab on your copy of the system disc before executing STOSCOPY. Full instructions are included along with this program.

Dump

This accessory allows you to edit the contents of any part of the ST's memory. Each byte of memory is displayed in both Ascii and hexadecimal formats. To edit a memory location move the cursor over the appropriate point and input your new data. When you have finished, press Return to enter the changes into memory. These changes can be reversed by pressing Undo.

Arrow keys	Move the cursor around the current screen.
Insert	Displays the last page of data.
Home	Displays the next page of data.
Enter	Enters any changes into memory.
Undo	Reverses the changes.

Note that the MENUS allow you to examine and change any of 16 possible memory banks associated with each of the four editible programs in memory.

Creating an accessory

The only major difference between a STOS Basic accessory and a normal program is in its ability to be called up using the Help menu. In fact, these accessories are really just a specialised form of the multiple programs I mentioned earlier. It's often useful for an accessory to be able to tell whether it is executing as an accessory or directly as a Basic program. This can be done with the ACCNB function.

ACCNB *(Get accessory number)*

ACCNB returns a value of zero if a program is not installed as an accessory, and a number between 4 and 15 if it is. This number represents the program number of the accessory.

Example:

```
new
10 ? accnb
20 wait key
```

Save this program as an accessory using the line:

```
save "acctest.acb"
```

Now type:

```
accnew
accload "acctest.acb"
```

If you run the program directly from the editor then the number zero will be printed. But if you call up the accessory named *acctest* from the Help menu, the number which is displayed will be equal to the function key you pressed + 4.

Now for a simple example of an accessory.

```
new
10 windopen 1,22,5,18,4,5
20 curs off
```

4 Sprite commands

STOS Basic allows you to move and animate up to 15 sprites at any one time. These sprites can represent anything from space ships to monsters, and can be created using a powerful sprite definer included as part of the STOS package. All sprite movements and animations occur completely independently of the rest of the system. This means that your program can be doing something totally different whilst the sprites are whizzing around on the screen regardless.

The Sprite Definer

STOS incorporates an extremely impressive sprite definition utility which allows you to quickly create large sets of sprites for use by your Basic programs. You can load this designer from the accessory disc with either:

```
load "sprite.acb":rem Load as a normal Basic program (Execute with RUN)
```

or

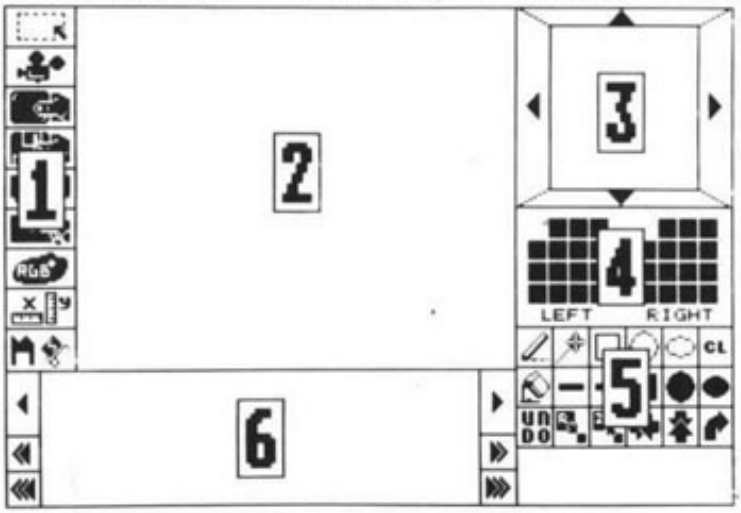
```
accnew:accload"sprite":rem Load as an accessory (Execute from HELP menu)
```

Because of the memory constraints on a standard 520 ST you should always remove all other STOS Basic accessories from the system before using ACCLOAD. Furthermore, it would also be a good idea to boot STOS Basic directly from the AUTO folder, as this will save you an additional 32k of memory.

It is important to note that designer runs in LOW resolution only. Don't panic if you're restricted to a mono monitor! A separate version of the package has been especially provided for you on the accessory disc – this will happily work in all three resolutions. Although this may seem a little less powerful than designer, it is still capable of generating some stunning effects, and indeed many of the example sprites on the disc were created using just this utility.

If you have enough available memory it's best to install the sprite editor as an accessory, as this enables you to access it instantly from within your STOS Basic program by pressing the <HELP><F1> keys.

On startup, designer automatically grabs any sprites which are currently employed by your program. You then simply remove the title screen with the left mouse button, and the sprite editor is ready for business.



4 Sprite commands

STOS Basic allows you to move and animate up to 15 sprites at any one time. These sprites can represent anything from space ships to monsters, and can be created using a powerful sprite definer included as part of the STOS package. All sprite movements and animations occur completely independently of the rest of the system. This means that your program can be doing something totally different whilst the sprites are whizzing around on the screen regardless.

The Sprite Definer

STOS incorporates an extremely impressive sprite definition utility which allows you to quickly create large sets of sprites for use by your Basic programs. You can load this designer from the accessory disc with either:

```
load "sprite.acb":rem Load as a normal Basic program (Execute with RUN)
```

or

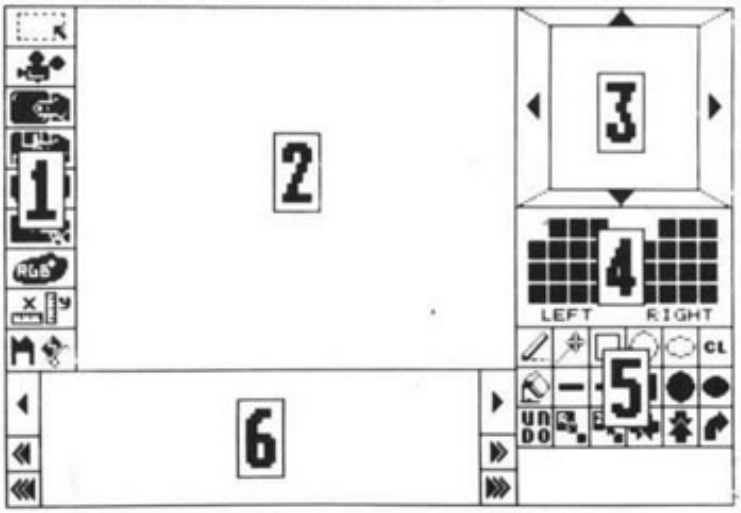
```
accnew:accload"sprite":rem Load as an accessory (Execute from HELP menu)
```

Because of the memory constraints on a standard 520 ST you should always remove all other STOS Basic accessories from the system before using ACCLOAD. Furthermore, it would also be a good idea to boot STOS Basic directly from the AUTO folder, as this will save you an additional 32k of memory.

It is important to note that designer runs in LOW resolution only. Don't panic if you're restricted to a mono monitor! A separate version of the package has been especially provided for you on the accessory disc – this will happily work in all three resolutions. Although this may seem a little less powerful than designer, it is still capable of generating some stunning effects, and indeed many of the example sprites on the disc were created using just this utility.

If you have enough available memory it's best to install the sprite editor as an accessory, as this enables you to access it instantly from within your STOS Basic program by pressing the <HELP><F1> keys.

On startup, designer automatically grabs any sprites which are currently employed by your program. You then simply remove the title screen with the left mouse button, and the sprite editor is ready for business.



At first glance the sprite designer may seem rather daunting. Once you have mastered the basic principles however. using it will quickly become second nature.

The screen can roughly be divided into six separate sections. These have been numbered from 1-6 in the above diagram.

Here is a breakdown of their various functions.

1 The system menu

The system menu contains nine icons which control the main features of the designer. Typical options available from this section are load/save. change size. and a clever facilty to allow you to design an animation sequence. These commands can be accessed directly from the screen by moving the mouse pointer over the appropriate icon and pressing the left button. A full list of the system icons can be found on page **64**, along with a detailed explanation of each function.

2 The drawing area

This is the area on the screen in which your sprite will be drawn. Points can be plotted at the current cursor position by pressing either the left or the right mouse buttons. As a default the right key is set to the background, and the left key to the colour white. You can change these colours whenever you like using a special Colour window.

3 The scroll zone

The scroll zone allows you to see the relative size of your sprite, and scroll it in all four directions. This scrolling can be activated at any time by clicking on one of four different icons which border the zone:

(Scrolls the sprite one pixel up)

(Scrolls the sprite one pixel down)

(Scrolls the sprite to the left)

(Scrolls the sprite to the right)

4 The colour window

This is divided up into two sets of 16 colours. One set of these colours is for the left mouse button, and the other is for the right. To select a a new colour for the mouse, you simply move the mouse pointer over the new colour and press the left button. Your current choice will now be highlighted on the screen.

5 The tools section

The tools area contains 18 different drawing icons. These include facilities to

create circles, ellipses and bars as easily as a single point. There's also an extremely useful undo feature which immediately reverses the effects of your last command.

You can choose one of these functions by simply clicking on the appropriate icon. The shape of the mouse pointer will now be changed accordingly to indicate the option you have selected. Most functions require you to first set the dimensions of an object before it can be drawn on the screen.

You normally specify the size of an item by keeping the left button pressed while moving the mouse. When you release this button, the object can be moved about with the mouse. You can now draw as many copies of the design on the screen as you wish by pressing the left button at any point in the drawing area. Incidentally, if you want to draw another object you can immediately reset the size back to zero with the right mouse button.

6 The Selection window

The selection window is used to display all the sprites which are currently installed in the ST's memory. Several of the system options use this window to allow you to choose one of a number of images which are currently held in the ST's memory. You can scroll through these sprites using the following icons:

(Smoothly moves the list back one place)

(Smoothly moves the list forward one place)

(Quickly moves the sprites backwards)

(Quickly moves the sprites forwards)

(Moves to the first sprite in the list)

(Moves to the last sprite in the list)

The tools icons

The tool icons provide you with a comprehensive set of drawing operations which make it extremely easy for you to design your own sprites.

(Plot a point)

In order to plot a point at the current mouse position, simply click on either the left or right mouse buttons. The colour of these points can be independently set from the colour window.

(Draw a line)

This draws a straight line in the drawing area using the colour assigned to the left mouse key. You first stretch the line to the length desired by pressing on the left button while moving the mouse. When you release this button, the line will be assigned directly to the pointer, and you can now draw any number of copies on the screen.

Incidentally, if you move the mouse outside the drawing area, the pointer reverts to an arrow, and can be used to access any of other commands without interferring with the current setting. This enables you to change the colour of the line you are defining directly from the colour window. When you move back to the drawing area, the cursor is immediately replaced by a line in the new colour.

As a general rule, all the drawing options can be employed using the following technique.

1. Set the size and shape of the object by pressing the left button at the same time as you move the mouse.

2. Release this button to assign the currently defined object to the mouse pointer.

3. Move the mouse to the position in the sprite where you wish your object to be placed and click on the left mouse button. You can now repeat this step several times to draw a number of copies of the object on the screen.

4. Remove the object from the mouse by pressing the right button.

(Draw a hollow box)

This draws a hollow box which can be expanded and contracted using the left mouse button as explained above.

(Draw a hollow circle)

Draws a hollow circle whose radius can be specified by holding on the left mouse button whilst moving the mouse.

(Draw a hollow ellipse)

Draws a hollow ellipse. The width of the ellipse can be specified by pressing the left button while the mouse is moved either left or right. Similarly, the height can be set by moving the mouse up or down.

(Erase definition)

The clear option erases the current drawing completely. As the effect of this command is permanent, you are always asked for confirmation before the sprite is erased. Note that this has no effect on any sprites which have been previously installed in the ST's memory.

(Fill an area)

Fill paints any hollow section of your sprite with the colour assigned to the left mouse button. To use this function, move the mouse inside the part of the drawing you wish to paint and press the left button.


and


(Choose fill pattern)

These options allow you to choose which of the many possible fill patterns will be used by any subsequent drawing operation. The current pattern is displayed in a small box positioned immediately below the TOOL icons.

(Choose the previous fill pattern from the box)

(Choose the next fill pattern from the box)

(Draw a filled bar)

Similar to box but draws a filled bar rather than a hollow box.

(Draw a filled circle)

This draws a filled circle which is defined in a similar manner to that used by circle.

(Draw a filled ellipse)

Draws a filled ellipse. See ellipse for more details.

(Undo the last change)

Undo is a very useful function indeed! This is because it enables you to instantly reverse the effect of your last drawing operation from the screen whenever necessary. Undo can be accessed either from the tools area, or directly from the keyboard using the <UNDO> key.

(Reduce sprite)

This function allows you to reduce the entire sprite into the top left hand corner of the screen. The magnitude of the reduction can be set using the left mouse button. **Warning**! Reduce is **not** the same as Change size. Instead of simply changing the definition of the sprite, reduce compresses the actual image. Some of the picture quality is therefore lost every time you perform this operation. Note that if you reduce a sprite and don't like the results you can easily return the sprite to its original size with <UNDO>.

(Zoom sprite)

Zoom expands the sprite up to twice its initial proportions. As with reduce the size of the zoom can be easily specified with the mouse. After the sprite has been expanded, you must always confirm the zoom by pressing the left button. Also note that you can use this option several times in sucession to enlarge the sprite to any size you wish. **Do not** confuse with change size.

(Reverse sprite)

Reverse mirrors the sprite from left to right.

(Invert sprite)

The invert icon flips the sprite from top to bottom

(Rotate sprite)

This rotates the sprite anti-clockwise in 90 degree steps. Note that rotate will only work if the width of your sprite is exactly the same as its height.

The system icons

The system icons control all the major features of the system, and allow you to specify a number of important attributes which define the appearance of your sprites.

I'll deal with these options in turn, starting from the top of the menu line and continuing to the bottom.

Cut and Paste

(Block menu)

The block icon gives you access to an impressive array of cut and paste operations. Here is a list of the powerful features supported by this command.

(Return to the main screen)

You can also click on the right mouse button to achieve the same effect.

(Block defined)

This option is highlighted if a section of the screen has been previously cut.

(Define a block)

You use this option to copy a section of the screen from one place to another. You first choose the area you wish to cut from the image by enclosing it with a rectangular box. Press the left button on the corner of this section and move the mouse cursor to specify its size. When you now release this button the block will be cut, and a copy stored in the ST's memory. If the erase option has been previously set, the original contents of the zone will be cleared from the screen using the background colour. You can then copy this block to any point on the screen with the mouse.

(Opaque toggle)

If this option is OFF then the background of the block will be transparent. Otherwise it will be OPAQUE.

(Cut and erase)

Erase informs the system that the source image will subsequently be erased from the screen immediately after a CUT operation is performed.

(Grab bottom right)

Grabs the block by its bottom right corner.

(Grab the upper left)

Grabs the block using its upper left corner

(Grab upper right)

Grabs the block using its upper right corner

(Grab bottom left)

Grabs the block using its bottom left corner.

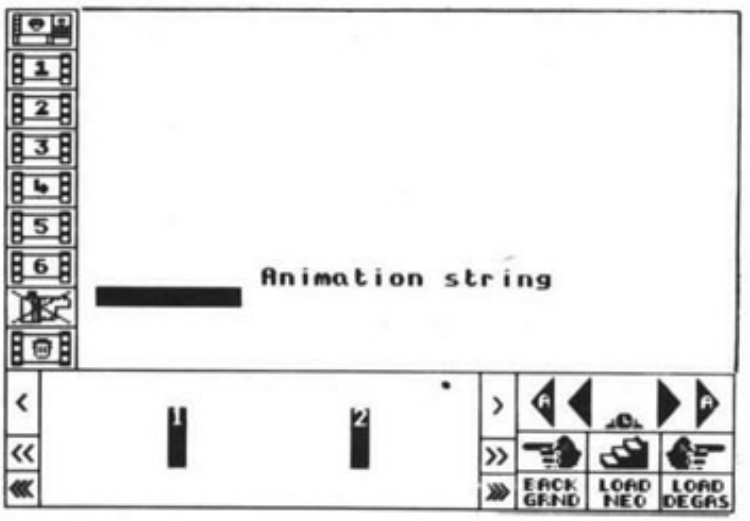
Note that all the usual features of the system such as Undo and Scroll also remain available within this mode.

Creating an Animation sequence

(Animate menu)

This option enables you to animate a sprite, and then play around with it until you are happy with the results. Just to make things easier, it automatically displays the exact string which would be used to achieve the same effect from the ANIM instruction.

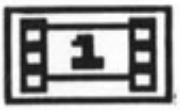
When you enter this mode, the following screen is displayed:



The first thing you notice about this screen is that the original systems icons have been completely replaced by the following list:

(Return to main menu)

Reverts back to main menu. Also executed by pressing the right mouse button.



(Animate 1)

Choose the First of six separate animation sequences.

(Animate 2)

Choose the Second animation (...and so on up to six)

(Erase film)

Erases the whole of the current animation.

(Delete frame)

Deletes a single frame from the animation.

In order to create your animation sequence, you first need to select the number of frames to be animated. This can be done by simply clicking on the appropriate sprite in the Selection window with the left mouse button. Your sprite will now be added to the current progression, and the string associated with it will be displayed on the screen. As a default the animation takes place at the centre of the drawing area. You can however move this display anywhere else you like on the screen using the mouse.

Controlling the Animation

The effect of the animation is controlled from a special dialogue box positioned to the immediate right of the selection window. At the top of the box is a line comprising of four arrows and a number. The number in the centre indicates the delay in 50ths of a second between the last image in the sequence and the next one you select. You can change this number up or down by clicking on the inner arrows.

You can also highlight any single animation string using the mouse cursor. The speed setting of this string will now be altered whenever you press the inner arrows, allowing you total control over the speed of each individual animation step.

The second set of arrows on the control panel change the speed of the animation as a whole. They do this by adding or subtracting one unit of time from all the animation strings you have defined. It is important to note that this option retains any differences between each of the separate stages.

Changing the direction

The second line of the dialogue box lets you change the direction of the animation, and also provides you with the ability to step through your animation a single frame at a time. There are three different options available from this section.

(Forward animation)

Executes the animation string from left to right.

(Reverse animation)

Executes the animation string from right to left.

(Step-by-step animation)

When this is set to ON, clicking on the mouse (while the pointer is outside the control panel) executes a single animation step.

Displaying a background screen

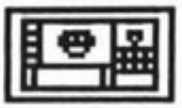
The final set of options enable you to load a screen in either Degas or Neochrome format into the background. This can now be displayed along with your animation using the BACKGRND icon. **Warning!** These screens **overwrite** any pictures you have loaded with the Grab image option.

Grabbing sprites from the disc

(Grab image)

This command enables you to grab sprites directly from a file in either Degas or Neochrome format.

There are seven possible options.



(Return to main menu)

Returns you back to the main menu

(Grab image)

Displays the current picture on the ST's screen. In order to grab a sprite from this picture you always need to follow the steps outlined below.

1. Define the size of your sprite by enclosing it with a hollow rectangular box. As you move the mouse with the left button held down, the dimensions of this box will expand and contract. When you release the button the dimensions of the sprite are set to the current size.

2. Move the box over the part of the image you wish to grab.

3. Grab the contents of this box into the sprite bank by pressing the left button.

(Grid on/off toggle)

When this toggle is ON the grab can only start on word boundaries. This helps when grabbing sprites that are snapped onto a boundary.

(Auto insert toggle)

If this option is ON the grabbed sprite will be transfered directly into the store.

(Grab from Neochrome picture)

Reads a Neochrome file off the disc. If the Get Palette option has been selected then the palette is loaded automatically along with the picture.

(Grab from Degas picture)

Loads a Degas file off the disc. If the Get Palette option has been selected then the palette is loaded automatically along with the picture.

(Get palette during grab)

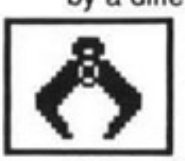
Loads the current palette of colours with the settings used by the new picture.

To exit from this mode click once on the right mouse button.

Grabbing a sprite from a program

(Grab from the program file)

This enables you to grab a sprite out of an program stored in a disc file. Unlike Grab image, this file doesn't have to be in any particular screen format at all. It can in fact, be anything from your favourite commercial game to a sprite file generated by a different editor.



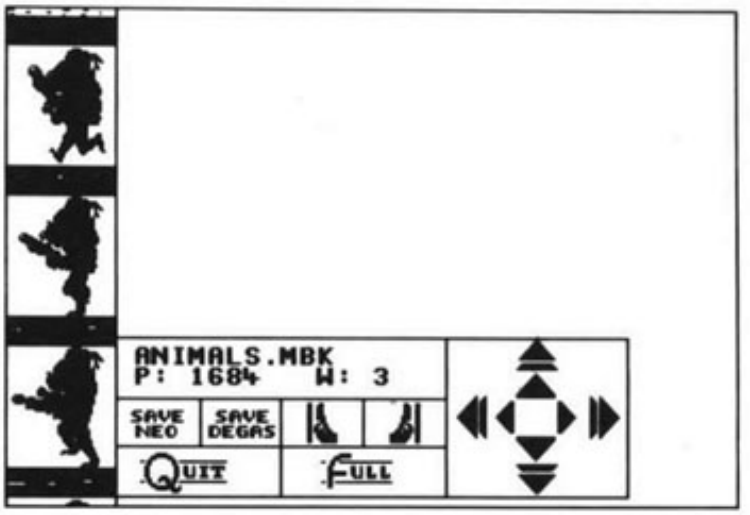
(Grab image)

Select this to grab a sprite from the loaded file.

(Select and grab from a file)

This erases the current screen and loads part of the file into the ST's memory. The contents of this file is now displayed in the form of a screen image.

At the bottom of the screen lies the main control panel.



As you can see, two numbers are displayed directly underneath the name of your file.

P:

This number indicates your position in the file. Note that since the designer loads each file in 16k chunks, there is no real limit to the size of the file you can inspect with this function.

W:

W denotes the current screen width, and can vary from 1 (very thin) to 20 (Full screen). The width can easily be changed by clicking on the icons situated just beneath the W. You can also redisplay the full screen with the Full icon. The width option is needed because different games store sprites in different formats. As a general rule, if the screen you are currently displaying looks like garbage, try altering its width – you could well be astonished at the results.

Searching through the file

On the right of the screen lies two sets of direction arrows which enable you to scroll through the file in search of some useful images.

The single arrows move the display through the file either a line (for the up/down), or a single byte (left/right) at a time. The four double arrows work in units of either 10 lines or 8 WORDS, depending on the direction of the motion.

Once you've found something interesting, you can save the entire screen using the Save Neochrome or Save Degas options.

You can also grab any individual sprite from this image. First press the right button to remove the control panel. Now select the sprite with the left button in the same way as with the grab image command.

Finally there is the Quit option. This returns you to the main menu without erasing the file you are inspecting. The next time you enter Grab programs, your current screen will be waiting for you at exactly the same point.

The FILE menu

(Disc file menu)

This is the menu which is used to save and load your sprites to the disc. These sprites are always stored in memory bank number 1. See RESERVE for more details.

(Use palette)

When this option is ON all files saved will have the current colour palette saved with them. Files loaded into the editor will change the current palette.

(Load a sprite file)

This loads a set of sprites from the disc. These are placed in bank 1 and replace any other sprites which were previously occupying this bank. Note that if you have selected the Palette option, then the palette used by the sprites will be loaded automatically by this function.

(Merge a sprite file)

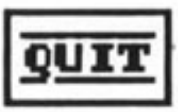
This command appends a sprite bank held on the disc to the one which is stored in memory. **Warning:** Merge only combines the sprites stored in LOW resolution. Like Load, the palette will be amended if you have set the Palette option to ON.

(Save)

SAVE saves the current contents of sprite bank 1 to the disc. **Warning:** Any sprites you wish to save **must** first be placed in the sprite bank with the Put Sprite option before this function is called – otherwise your data will be **lost.**

(Save as)

Saves your sprites under a new filename.



(Quit)

Leaves the sprite designer, losing any sprites you have defined.

(Quit & grab)

This option only makes sense if the designer has been executed as an accessory. Quit & Grab then leaves the definer, and copies the sprites you have defined straight into the current program.

Changing the Hot Spot

(Hot Spot menu)

Each sprite is manipulated on the screen using a special point called the Hot Spot. This can be changed to anywhere inside the sprite using the Hot Spot Menu. To see the current setting, move the mouse into the drawing area. The hot spot will now flash continually on the screen.

In order to make life easier for you, a number of commonly-used settings have been assigned to the icons.

(Upper left)

Set Hot spot to the upper left hand corner of the sprite.

(Upper middle)

Set hot spot to the middle of the upper line of the sprite.

(Upper right)

Set hot spot to upper right corner.

(Bottom left)

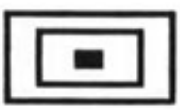
Bottom left corner.

(Bottom middle)

Middle of bottom line.

(Bottom right)

Bottom right corner.



(Centre)

This positions the Hot Spot to the centre. One useful side effect of this is to indicate the precise centre of the sprite. By scrolling the sprite using the scroll window, you can therefore use this feature to neatly arrange your sprite on the screen.

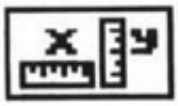
Changing the palette

This can be achieved with the RGB option will allows you to specify one of 512 possible shades for each of the 16 available colours.

(Alter palette)

To use this feature, first click on the colour you wish to change in the LEFT colour window. You can also select the colour by clicking on any individual point in the drawing area. Now move the Red/Green/Blue sliders to set this colour to a specific value. If you wish to reverse the last colour setting you can as usual, click on the UNDO option. Finally press the right mouse key to return back to the main menu.

Changing the size of the sprite



(Set X and Y menu)

STOS Basic allows you to use sprites ranging from 16x2 to 64x64 pixels in size. As a default the size is set to 32x32 but this can be changed at any time from the SET X and Y menu. When you call this option the current size is displayed on the screen. You can now alter this setting using the scroll window. Note that the width of the sprite can only be altered in 16 pixel steps. You should also remember that the HOT SPOT of the sprite is always reset back to the top left corner of the screen, whenever the SET X and Y function is called.

(Squeeze sprite)

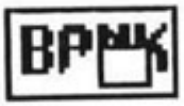
If you press on this menu selection the sprite in the edit window will be moved into the top left-hand corner. This frees the surrounding space and allows you to shorten the width and height of the spite, thus achieving the smallest size possible.

Placing a sprite into the bank

After you created one of your sprites you must always remember to place it into the sprite bank. This can be done using the store sprite menu.

(Store menu)

Here is a list of the various options.



(Erase bank)

Erases the entire Bank. Since erase is **very** dangerous indeed, you are always asked for confirmation before this function is executed.

(Delete sprite)

Deletes the sprite picked from the selection window. Note this option is permanent and cannot be undone!

(Insert sprite)

INS inserts the sprite at the current slot by shifting all the sprites one place to the right. This makes a space for the new definition in the the memory bank.

(Put sprite)

This copies the sprite you are currently editing into the sprite displayed in the centre of the selection window. In order to avoid overwriting your existing sprites, you should position the first empty slot at the middle of the window before use. **Warning!** This option **erases** any data already stored in the destination sprite.

(Get sprite)

Edits the sprite you have chosen with the selection window.

To save a great deal of menu switching we have included some functions that allow you to put and get sprites with super speed. When editing a sprite you can place it into the store by pressing the down arrow key twice, this is the same as using the put sprite option from the store menu. To get a sprite from the store just press the up arrow key twice.

For real speed you can put the sprite in the editor and then get the next sprite form the store just by pressing the right arrow key. If you press the left arrow key then the edit sprite will be stored and the previous sprite will be loaded into the drawing area.

Using the Sprite designer

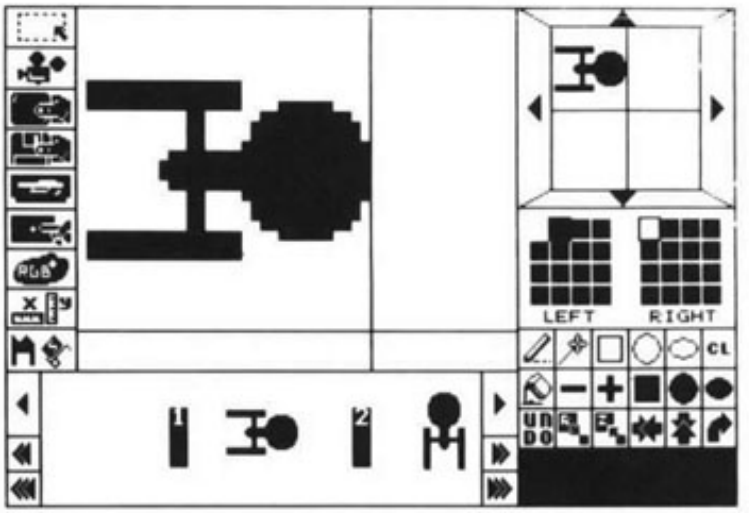
So far, we've only concentrated on theory. In this section, I'll be showing you how the sprite designer can be utilized to draw an actual set of sprites for use in one of your own programs.

Before we can do anything, we first need to load the sprite editor into memory. Type the line:

accnew:accload "SPRITE"

Now enter the designer using <HELP><F1>

As an example, we'll be creating a sprite representing a certain well-known spaceship. Here is a picture of the type of effect we will be aiming for:



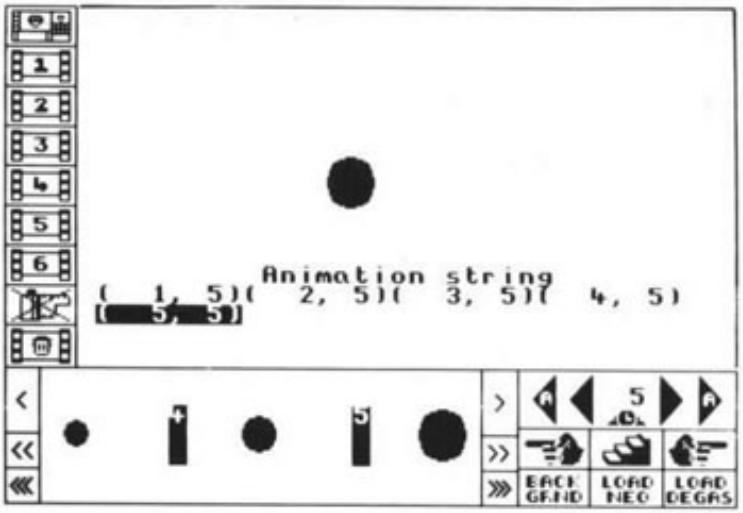
Drawing an image

We'll start off by selecting the colour of our new sprite. Move the mouse over the left colour window and choose a nice bright shade for the sprite by pressing the left button over one of the colours.

We will now draw the large disc which forms a major part of the ship. Click on the disc option from the tools menu to set the pen to a filled circle. Move the pointer into the drawing area and press the left button as you pull the mouse to the right. This generates an expanding disc on the screen. When the disc is about a third of the size of the drawing area, release the button to assign it to the mouse. We can now place this circle in the centre right of the sprite and fix it into position with the left mouse button.

Now for the so-called primary hull. For this section we'll need to draw a filled bar from the middle of the disc to the edge of the screen. Select the bar option and move the mouse to the centre of the disc. Now expand the bar by holding onto the left button while you move the mouse to the left. Release the button when the bar has reached a reasonable size. We can then push the hull into position and click on the mouse to set it in place.

Finally, we will produce the two outriggers which are so distinctive of this type of space ship. First erase the last bar with the RIGHT mouse button. Now shift the pointer to the top of the sprite and draw a thin bar passing straight through the primary hull. This forms a strut which will connect the two outriggers to the main part of the ship. We can then move the mouse to the top left of the sprite and generate a thin horizontal bar. Position this in the centre of the strut and click the left button, and repeat this process at the equivalent point at the bottom of the sprite. You should now be looking at a picture similar to the one I showed you earlier.



Now try moving the mouse pointer around on the display area and clicking on the left button. As you can see, the entire animation moves immediately to the new position.

We will manipulate our animation by moving the mouse to the control window and clicking on the left and right "A" arrows. These change the speed of the entire sequence. We can also alter the speed of just one of the images. Let's choose an animation to be affected by moving the pointer over an appropriate string. We can then change the speed of this step by selecting any of the inner most arrows.

Let's invert the animation sequence. If we select the reverse icon with the left mouse button, the images will now be displayed in revese order and the circle will appear to contract into nothing.

We can also display the animation against a background screen stored on the disc. This can be done using the load Degas icon from the control panel.

If we place the STOS system disc in the drive we can now load the title screen (in PIC.PI1) from the STOS folder. To display the new screen alongside our animation sequence we then click on the BACKGRND icon. We can then return to the command screen by pressing the right mouse button.

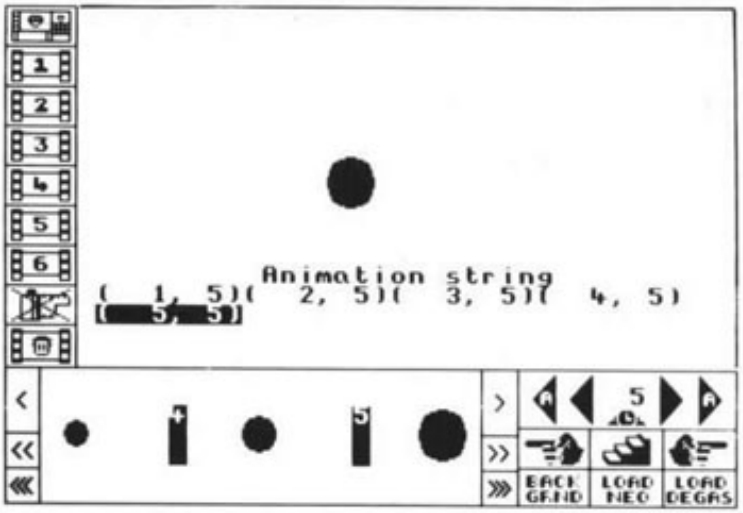
Finally, we should always end our session by making a note of the animation string on a scrap of paper. This will be needed when we wish to recreate our sequence using the STOS Basic ANIM instruction. We can now press the right mouse button to return to the main menu, and save our sequence to the disc using the save option from the file menu.

The multiple-mode sprite definer

For the users who wish to design sprites in medium and high resolutions, we have included a breakdown of the sprite editor which can operate in all three modes.

This can be found in the file SPRITE2.ACB on the accessory disc.

In many respects SPRITE.ACB is just a simpler version of SPRITE, and indeed many of the basic techniques I discussed earlier will also apply equally well to either of these two programs. One minor advantage of SPRITE2.ACB is that is uses considerably less memory than the more powerful SPRITE program.



Now try moving the mouse pointer around on the display area and clicking on the left button. As you can see, the entire animation moves immediately to the new position.

We will manipulate our animation by moving the mouse to the control window and clicking on the left and right "A" arrows. These change the speed of the entire sequence. We can also alter the speed of just one of the images. Let's choose an animation to be affected by moving the pointer over an appropriate string. We can then change the speed of this step by selecting any of the inner most arrows.

Let's invert the animation sequence. If we select the reverse icon with the left mouse button, the images will now be displayed in revese order and the circle will appear to contract into nothing.

We can also display the animation against a background screen stored on the disc. This can be done using the load Degas icon from the control panel.

If we place the STOS system disc in the drive we can now load the title screen (in PIC.PI1) from the STOS folder. To display the new screen alongside our animation sequence we then click on the BACKGRND icon. We can then return to the command screen by pressing the right mouse button.

Finally, we should always end our session by making a note of the animation string on a scrap of paper. This will be needed when we wish to recreate our sequence using the STOS Basic ANIM instruction. We can now press the right mouse button to return to the main menu, and save our sequence to the disc using the save option from the file menu.

The multiple-mode sprite definer

For the users who wish to design sprites in medium and high resolutions, we have included a breakdown of the sprite editor which can operate in all three modes.

This can be found in the file SPRITE2.ACB on the accessory disc.

In many respects SPRITE.ACB is just a simpler version of SPRITE, and indeed many of the basic techniques I discussed earlier will also apply equally well to either of these two programs. One minor advantage of SPRITE2.ACB is that is uses considerably less memory than the more powerful SPRITE program.

Another benefit is that the accessory will happily allow you to create files containing sprites in each of the three resolutions simultaneously. This is especially useful when designing new pointers for the mouse.

You can load SPRITE2.ACB at any time with the line:

```
accnew:accload "SPRITE2.ACB
```

On startup the screen is split into six seperate windows

- **The information line:** This is placed at the top of the screen, just underneath the menus. It is used to display any relevant information such as the colour of the current pen or the size of the sprite.
- **The RGB Window:** Click on one of the letters R/G/B to change the colour setting used by the mouse for all future drawing operations.
- **The scroll window:** This is utilised by the SCROLL option ot scroll the sprite in all four directions.
- **The pattern window:** Holds a copy of the current fill pattern. You can change it by repeatedly clicking on this window with the left and right mouse buttons to page through the various possibilities.
- **The sprite display:** This displays a full-sized copy of the sprite you are editing.
- **The drawing window:** The drawing window is used to edit your sprite. To plot a point at the current pointer position simply click on the left button. The right mouse button can also be used in a similar fashion to delete a point from the sprite

Here is a breakdown of the various menu options available from this program.

STOS

Sprites

Displays a title screen. Click the mouse to remove.

Quit

Exits from the sprite definer, losing all of your current sprite definitions.

Quit and Grab

Exits from the definer and incorporates any new definitions into your current program. This option only works if the definer has been executed as an accessory.

File

Load Sprite Bank

Loads a file containing a list of sprites into bank number one. These can be edited using the get sprite option.

Save Sprite Bank

Saves all the sprites you have defined into a new file on the disc.

Save as...

Saves the bank using a different filename than the one it was originally loaded from.

BANK

Grab from program

Grabs any sprites used by your current program fro subsequent editing by the definer. Obviously this option only applies if you have loaded the definer as an accessory.

SPRITE

Put Sprite

Puts the current sprite into a particular slot and replaces any of the original contents.

Insert Sprite

Inserts the sprite you are editing into bank 1, without overwriting any of the existing images.

Get Sprite

Gets a sprite out of the memory bank to be edited.

Erase Sprite

Erases one of the sprites from the bank.

You can select the sprite used by these functions by clicing the left button over the appropriate image in the drawing window. These sprites are displayed in groups of nine. To page through the entire set, simply click on the NEXT and previous boxes below this window.

Move Sprite

This allows you to assign one of the sprites to the mouse and then see how it looks when you move it around on the screen

Cinema

The Cinema option enables you to animate your sprites from within the definer. To choose the sprites which will make up your animation sequence, simply click on the appropriate images in the drawing area. Then click on the left mouse anywhere outside this window to start the animation running. You can now change the speed of the animation with the + or – keys.

Previous cinema

Restarts the last animation sequence you defined from the point you left off.

Get from DEGAS
Get from NEO

Grab a sprite from a screen stored on the disc in DEGAS or NEOCHROME format. After you have chosen the file with the file selector, you are then presented with a list of the currently defined sprites in the bank. Select the one you wish to load using the left mouse button. Note that the dimensions of this sprite determine the final size of the image which will be grabbed.

The new screen is now displayed and you can grab the image which is underneath the mouse cursor by pressing the left mouse button. After you have finished you can return to the editor by clicing on the right mouse button.

FIX mask

This allows you to select the mask colour used as the transparent index.

Fix Hot Point

Click the left button on the appropriate point to set the hot spot of the sprite. The current spot can be seen flashing on the screen.

Fix X and Y Size

This allows you to change the dimensions of the sprite. Click on the scroll arrows to alter the size.

TOOLS

Erase

Erases the currently edited sprite. Does **not** affect any sprites stored in the bank.

Mirror

Reverses the sprite from left to right.

Flip

Reverses the sprite from top to bottom.

Scroll

Scrolls the sprite. Click on the arrow keys to scroll the sprite in any direction.

Paint

Whenever you subsquently click the mouse in an enclosed area in the sprite, this will be filled with the current fill colour using the pattern you have selected from the FILL window. Click on DRAW to revert the editor back to normal.

Palette

This provides you with a list of the colours available for your use. Click on a colour to assign it to the current pen.

The SPRITE command

After we have drawn our sprites with the sprite definer, we will obviously need some way of displaying them on the screen. This can be done using the SPRITE instruction.

SPRITE *(Displays a sprite on the screen)*

SPRITE n,x,y,p

This displays sprite number *n* on the screen at coordinates *x* and *y*.

n is the number of the sprite, which can range from 1 to 15. It is this number which will be used to identify the sprite in any subsequent calls to the MOVE and ANIM instructions.

x and *y* are the coordinates of the point on the screen where the sprite is to be drawn. Unlike normal screen coordinates, these can take NEGATIVE values. The *x* coordinate can vary from -640 to +1280, and the *y* coordinate from -400 to +800. This allows you to move the sprite off screen without causing an error.

p specifies which of the images in bank 1 is to be used for a particular sprite. The only limit to the number of these images is the amount of available memory.

Each sprite has an invisible handle through which it can be manipulated, called a Hot Spot. Whenever we draw a sprite, we always specify its coordinates in terms of the position of this point on the screen. As a default, the hot spot is always set to the top left hand corner of the image, but this can readily be changed using a special option from the Sprite definer accessory.

Examples:

A number of example sprites have been placed on the accessory disc for your use. You can load one of these sets using the LOAD instruction like so:

```
load "fontset.mbk"
```

This loads a collection of sprites which depict the various letters of the alphabet.

Now let's display some of these sprites on the screen.

```
mode 0:rem These sprites are designed for low resolution flash off
palette 0,$777,$444

sprite 1,100,100,6:rem Displays a 1 character at 100,100 as sprite 1
sprite 2,10,50,6:rem Displays another sprite with the same image
sprite 1,100,100,7:rem Change sprite 1 from a 1 to a 2
sprite 3,-10,100,5:rem Demonstrates the use of negative coordinates
```

It is important to realise that the sprite command effectively does two separate

things: Not only does it draw a sprite on the screen, but it also determines which image will be associated with each of the 15 sprite numbers. You must therefore always use this instruction BEFORE moving or animating a sprite.

Moving a sprite

Any of the STOS Basic sprites can be moved across the screen using interrupts, without affecting the execution of your Basic program in the slightest. The command which enables you to do this is very powerful indeed and is called, quite simply, MOVE. The MOVE instruction

This allows you to assign a complicated series of movements to a sprite, which will then be executed automatically by STOS Basic every 50th of a second (70th for high resolution). There are two main versions of this command, one for horizontal motions, and another for vertical movements. These can be combined to produce intricate patterns on the screen. Since the two instructions are otherwise identical, we will concentrate on the MOVE X command first, and then explain any significant differences between it and MOVE Y.

MOVE X *(Move a sprite horizontally)*

MOVE X n,m$

This defines a list of horizontal movements which will be subsequently performed by sprite number *n*. *n* can range from 1-15 and refers to the number of a sprite you have previously installed using SPRITE.

m$ contains a sequence of commands which together determine both the speed and direction of the sprite.

Each of these instructions is split into three separate components.

SPEED

This stipulates the delay in 50ths of a second between each successive sprite movement. The speed can vary from 1 (very fast) to 32767 (incredibly slow)

STEP

The STEP size specifies how many pixels the sprite will be moved in each operation. If this step is positive the sprite will move to the right, and if it is negative to the left. The apparent speed of the sprite depends on a combination of the speed and step. Large displacements coupled with a moderate speed will move the sprite quickly but jerkily across the screen. Similarly, a small step size combined with a high speed will also move the sprite very fast, but the motion will be much smoother. The fastest speeds can be obtained with a displacements of about 10 (or -10).

COUNT

This designates the number of steps which will be completed in a single movement. Possible values range from 0 to 32767. If you use a COUNT of 0, the motion will be repeated indefinitely.

These three elements are placed into the movement string using the following format: (speed,step,count)

Here is a simple example which should make this a little clearer. Load a set of sprites from the accessory disc with:

```
load "fontset.mbk"
```

Now define sprite 1 using the SPRITE instruction like so:

```
sprite 1,10,100,1
```

We can move this sprite with MOVE X:

```
move x 1,"(1,3,50)"
```

When we execute the above command, we find to our surprise that nothing happens. This is because we need to first initiate the motion using a special MOVE ON instruction.

```
move on
```

The sprite now progresses steadily across the screen. We can combine any number of these individual movements into a single MOVE command. They will then be executed in turn, one after another.

Example:

```
move x 1,"(1,1,100)(1,-1,100)"
move on
```

This moves the sprite from left to right, and back again.

There are also a couple of other directives available for our use. The most important of these extensions is the L instruction (for loop), which jumps back to the start of the list and reruns the entire sequence again from the beginning.

Example:

```
sprite 1,10,100,5:rem Define Sprite 5
move x 1,"(1,5,60)(1,-5,60)L"
move on
```

Another useful option is the E command which stops the sprite whenever it reaches a specific position on the screen.

Example:

```
sprite 1,10,100,5
move x 1,"(1,5,30)E100"
move on
```

The most common use of this instruction is to halt a sprite which has been defined with a count of zero at a particular point. The following example illustrates this technique.

```
sprite 1,10,100,5
move x 1,"(1,5,0)E200"
move on
```

Note that these endpoints will only work if the x coordinate of the sprite exactly reaches the value you orginally designated in the instruction. If this increment is badly chosen, the sprite will leap past the endpoint in a single step, and the test will therefore always fail.

Incidentally, you can also use an endpoint in conjunction with the L command. This has the effect of stopping the sprite and then executing the series of movements again from the start.

Example:

```
sprite 1,10,100,5
move x 1,"(1,5,30)L100"
move on
```

In the example above, the ending condition was pretty useless, because the motion immediately resumes from the point it had reached when the sequence was terminated. But you can also add an optional starting position to the movement. This returns the sprite back to its original location, and therefore allows you to loop the sprite repeatedly through a precise section of the screen. Here is an example of this function in action:

```
sprite 1,-10,100,1:rem Defines sprite 1 off screen
move x 1,"100(1,1,0)L200"
move on
```

The sprite now starts from 10,100, and slowly progresses to location 200,100 before looping back to 10,100.

See MOVE ON, MOVE Y, MOVE FREEZE, MOVON, ANIM, SPRITE, UPDATE

MOVE Y *(Move a sprite vertically)*

MOVE Y n,m$

This instruction complements the MOVE X command by enabling you to move a sprite through a complex series of vertical manoeuvers. As before, *n* refers to the number of a sprite you have installed using SPRITE, and ranges between 1-15.

m$ holds the movement string. This uses an identical format to MOVE X, except that positive displacements now correspond to a downward motion, and negative steps to an upward movement.

Examples:

```
load "fontset.mbk":rem Load sprites from accessory disc
sprite 1,100,10,5:rem Install sprite
move y 1,"10(1,1,180)L":rem Loop sprite from 10,10 to 190,10 continually

sprite 1,100,100,1
move y 1,"(1,4,25)(1,-4,25)":Rem moves sprite up and down
```

See MOVE X, MOVE ON, ANIM, SPRITE

Combining horizontal and vertical movements

Any list of horizontal and vertical movements may be combined with ease. All you need to do is to split the movement into separate horizontal and vertical

components, and then assign these to individual MOVE X and MOVE Y instructions. Here are a couple of simple examples which illustrate this process.

```
new
load "fontset.mbk":rem From accessory disc
sprite 1,0,0,22
move x 1,"(1,4,79)(1,-4,79)L"
move y 1,"(1,4,49)(1,-4,49)L"
move on
```

Now for a slightly larger example:

```
new
load "fontset.mbk"
5 rem Exploding Title
10 cls : click off
20 for I=1 to 10
30 read C : sprite I,I*16+80,100,C:rem Install sprites in centre of screen
35 rem Set alternate characters moving in different vertical directions
40 if I mod 2=0 then V$="(1,-2,0)" else V$="(1,2,0)"
45 rem Set left half moving left and right half moving right
50 if I<6 then H$="(1,-2,0)" else H$="(1,2,0)"
55 rem Set up Vertical and Horizontal components
60 move x I,H$ : move y I,V$
70 next I
80 wait key : boom : move on: Rem Wait for a keypress and move sprites 85 rem
Image Numbers of Sprites which make up title
90 data 40,41,36,40,18,23,22,40,30,24
```

MOVE ON/OFF *(Start/stop sprite movements)*

MOVE ON/OFF [n]

Before any sprite movements you have defined by the MOVE X and MOVE Y commands will be performed, they need to be initiated with this instruction. The optional expression *n*, refers to a number from 1-15 which indicates a single sprite you wish to move. If it is ommitted then all the movement sequences you have currently assigned, will be activated simultaneously.

Similarly, MOVE OFF kills the movements of the sprites in exactly the same way. Do **not** confuse MOVE ON with the MOVON function.

See MOVE X, MOVE Y, OFF

MOVE FREEZE *(Temporarily suspend sprite movements)*

MOVE FREEZE [n]

This command can be used to temporarily halt some or all of the sprites which are currently moving. These can be restarted again using MOVE ON. The value *n* is optional and specifies the number of a single sprite you wish to freeze.

Example:

```
load "fontset.mbk":rem From accessory disc
sprite 1,0,0,1
move x 1,"(1,4,64)(1,-4,64)L"
```

```
move on
move freeze
move on
```

=MOVON *(Return sprite state)*

x=MOVON(n)

This function returns a non zero number if sprite number *n* is currently in motion and 0 (FALSE) if it is stationary.

Example:

```
load "fontset.mbk":rem From accessory disc
move x 1,"(1,4,0)":menu on
print movon(1)
move off
print movon(1)
```

Do **not** confuse with the MOVE ON command.

=X SPRITE *(Get X coordinate of sprite)*

x1=X SPRITE(n)

Returns the current X coordinate of sprite *n*. This command is frequently used as a way of detecting whether a sprite has collided with the edge of the ST's screen.

Example:

```
load "fontset.mbk"
sprite 1,0,40,1
move x 1,"10(1,1,0)L320"
move on
for i=1 to 100:locate 0,0:print x sprite(1):next i
```

See also Y SPRITE, X MOUSE, Y MOUSE

=Y SPRITE *(Get Y coordinate of sprite)*

y1=YSPRITE(n)

This is very similar to the X SPRITE instruction, except for the fact that it returns its Y coordinate rather than the X coordinate. As usual, *n* refers to the number of the sprite and can range from 1-15. This command is often utilised to check whether a missile has passed off the top or bottom of the screen.

Example:

```
load "fontset.mbk"
sprite 2,0,0,35
move y 2,"0(1,1,0)L200"
move on
for i=1 to 100:locate 0,0:print y sprite(2):next i
```

A further example of this function can be found in the section on collision.

See also X SPRITE, X MOUSE, Y MOUSE

LIMIT SPRITE *(Limits sprite to a specific area)*

LIMIT SPRITE x1,y1 TO x2,y2

Defines the area of the screen on which the sprites will be displayed. Whenever they move outside this area, they will dissapear from the screen. Note that unlike LIMIT MOUSE, this command does NOT limit the actual movements of the sprites, only their visibility.

x1 and *y1* denote the top left corner of the zone, and *x2,y2* indicate the point diagonally opposite. All the X coordinates used in this command are automatically rounded down to their nearest multiple of 16.

Example:

```
load "fontset.mbk"
sprite 1,0,0,1
move x 1,"0(1,1,0)L320"
move y 1,"0(1,1,0)L200"
move on
limit sprite 100,50 TO 200,150
```

In order to return the sprites to normal, simply enter a LIMIT SPRITE command with no parameters like so:

```
limit sprite
```

See LIMIT MOUSE, CLIP

Animation

STOS Basic supplies you with a simple command called ANIM which can be readily used to animate your sprites. This can be used to produce a wide range of effects from a walking gorilla to an impressive explosion.

ANIM *(Animate a sprite)*

ANIM n,a$

This enables you to page through a chain of sprite images one after another. This sequence will be executed at the same time as your sprite is being displayed, even if it is also being moved using MOVE.

n refers to the number of the sprite to be animated, and *a$* to a list of animation commands to be carried out.

The string *a$* contains the set of instructions to the ANIM command. Each operation is split into two separate components enclosed between brackets.

IMAGE

This is the image number of the sprite to be displayed during each step of the animation.

DELAY

Specifies the amount of time the image will be held on the screen before the next image is displayed. This delay is input in units of a 50th of a second (70th for monochrome systems).

Here is a typical example of how this instruction works in practice.

```
anim 1,"(1,10)(2,10)"
```

This would display image number 1 for 10/50 or a 1/5 of a second, and then flick to image number 2.

Just as with the MOVE instruction, there's also an L directive which enables you to repeat these animations.

So we could repeat the above animation continually with:

```
anim 1,"(1,10)(2,10)L"
```

Now for a real example of the ANIM instruction. We'll use some of the sequences utilized by Zoltar for this purpose. Before we can play around with these sprites, we first need to grab them out of the game. The easiest way we can achieve this involves a number of separate steps. We start off by loading Zoltar from the Game disc with:

```
load "\zoltar\zoltar.bas"
```

We then place a fresh disc in the drive, and save the sprite bank in a separate file like so:

```
save "zsprites.mbk",1
```

Finally, we simply erase Zoltar from memory and reload the sprites with:

```
new
load "zsprites.mbk"
```

These sprites can now be accessed from within any of our example programs. To list the images which are currently available, type the following small routine:

```
10 mode 0 : cls : flash off
20 palette $0,$777,$3,$4,$17,$770,$530,$400,$555,$333,$111,$734,
$715,$706,$707,$770
30 for i=1 to 30:sprite 1,100,100,i:print i:wait key:next i
```

Note that the palette command in line 20 was discovered by searching through Zoltar with:

```
search "palette $"
```

If you run this program you will see that images 14 to 18 form a rather nice explosion. Let's animate this by replacing line 30 with:

```
120 sprite 3,100,100,14:anim 3,"(14,2)
(15,2)(16,2)(17,2)(18,2)" : anim on
```

We can observe this sequence more clearly if we add an L instruction to repeat the animation like so:

```
120 sprite 3,100,100,14:anim 3,"(14,2) (15,2)(16,2)(17,2)(18,2)L" : anim on
```

Note this large line number! This is to allow us to expand our program later.

Another interesting arrangement can be created using the images 2 and 3 which combine to produce one of Zoltar's wiggling missiles.

Animate this with:

```
30 sprite 1,160,198,2:anim 1,"(2,1)(3,1)L":anim on
```

and move it up the screen using:

```
40 move y 1,"196(1,-4,50)I" : move on
```

We'll now have a brief look at the sprites used to make up the spaceships. These are composed of groups of three sprites starting from image 19.

Let's add one of these ships to our current program. Type the lines:

```
50 sprite 2,0,40,9 : anim 2,"(19,4)(20,4)(21,4)L"
60 move x 2,"(1,4,80)(1,-4,80)I" : move on 2:anim on
```

When you run this program, the missile fires and the ship moves from left to right. We'll be modifying this program later in the section on collision, so it's a good idea to save it on a separate disc with a line like:

```
save "ship.bas"
```

ANIM ON/OFF *(Start an animation)*

ANIM ON/OFF[n]

Used to activate a series of animations defined using the ANIM command. *n* denotes the number of an individual sprite to be animated. If it is omitted then all the animation sequences you have created will be initiated at the same time.

ANIM OFF [n] stops one or all of the animations begun by ANIM ON.

ANIM FREEZE *(Freeze an animation)*

ANIM FREEZE [n]

This command temporarily pauses the current animations on the screen. If the optional *n* is included, only a single animation sequence will be suspended. Otherwise all the animations will be frozen. These can be restarted again with the ANIM ON instruction.

Controlling the sprite using the mouse

The easiest way to give the user control of a sprite is to assign the sprite to the mouse pointer with the CHANGE MOUSE command. We can then determine both the position and status of this mouse from within our program using the X MOUSE, Y MOUSE, and MOUSE KEY instructions.

CHANGE MOUSE *(Change the shape of the mouse pointer)*

CHANGE MOUSE m

This allows you to completely redesign the shape of the mouse at any time. Three forms are already installed into the system as a default, and are given the numbers 1 through 3. Here is a list of the various options:

m	Shape
1	Arrow. (Default)
2	Pointing Hand
3	Clock

If you specify a value of *m* greater than 3, this is assumed to refer to an image stored in the sprite bank. The number of this image is determined using the expression I=m-3. So image number one would be installed by a value of four, and image two would be signified by a five.

Here are a few simple examples. Load the sprites from the file fontset on the accessory disc.

```
load "fontset.mbk"
```

and assign image 0 to the mouse with:

```
change mouse 8
```

Similarly we can set the mouse to a capital S with the line:

```
change mouse 43
```

Another powerful option is to change the default definitions for the mouse which are stored on the disc. These can be found in the file /STOS/MOUSE.SPR on the systems disc.

You can replace these with another set like this:

- Define three sets of sprites, for EACH resolution. If you only want to affect one resolution, it's best to modify the sprites in SPRDEMO.MBK (from the accessory disc), as this already contains a bank of sprites in the correct format.
- Load these sprites into bank 1 using either LOAD or the QUIT and GRAB options from the SPRITE definer.
- Place a copy of the STOS Basic system disc in the drive. DO NOT USE THE ORIGINAL SYSTEMS DISC FOR THIS PURPOSE! Now type:

```
bsave "\stos\mouse.spr",start(1) to start(1)+length(1)
```

Whenever you subsequently load STOS Basic, the new mouse pointers will now be automatically utilized by the system.

See also HIDE, SHOW, X MOUSE, Y MOUSE, MOUSEKEY, LIMIT MOUSE

=X MOUSE *(Get the X coordinate of the mouse pointer)*

x1=X MOUSE

This function returns the current X coordinate of the mouse pointer.

Example:

```
new
10 home
20 print x mouse
30 wait vbl:rem Stop print interfering with mouse pointer
40 if inkey$="" then 20:rem Wait for keypress from keyboard
```

=Y MOUSE *(Gets the Y coordinate of the mouse pointer)*

y1=Y MOUSE

This function simply returns the current Y coordinate of the mouse pointer.

Example:

```
new
10 home
20 print y mouse
30 wait vbl:rem Stop print interfering with mouse pointer
40 if inkey$="" then 20:rem Wait for keypress from keyboard
```

=MOUSE KEY *(Get status of mouse keys)*

k=MOUSE KEY

Enables you to quickly test whether one or both of the mouse buttons have been pressed. It returns one of the following four numbers depending on the current state of the keys.

Value	Meaning
0	If no button has been pressed
1	left button pressed
2	right button pressed
3	both buttons pressed

Example:

```
10 if mouse key = 1 then print "Left button"
20 if mouse key = 2 then print "Right button"
30 if mouse key = 3 then print "Left and Right button"
40 goto 10
```

See X MOUSE, Y MOUSE

LIMIT MOUSE *(Limit mouse to a section of the screen)*

LIMIT MOUSE x1,y1 TO x2,y2

Restricts the mouse to the rectangular area defined by the coordinates (x1,y1) and (x2,y2). *x1,y1* denotes the top left hand corner of this box and *x2,y2* to the point diagonally opposite. Note that LIMIT MOUSE always repositions the mouse

pointer at the centre of the box. Also, unlike LIMIT SPRITE, the mouse is completely trapped inside this zone and cannot be moved anywhere else in the screen.

Example:

```
limit mouse 50,50 to 250,150
```

In order to restore the mouse to normal, simply use the instruction with no parameters like this:

```
limit mouse
```

HIDE *(Remove mouse pointer from the screen)*

This command permits you to remove the mouse pointer from the screen at any time. A count of the number of occasions you have called this function is automatically kept by the system. This number needs to be matched by an equal number of SHOW instructions before the mouse will be returned for your use.

There's another version of this instruction which can be accessed with HIDE ON. This ignores the count completely and ALWAYS hides the mouse. Note that HIDE only makes the mouse pointer invisible. It does NOT deactivate it fully. You can therefore readily use the X MOUSE and Y MOUSE functions to read position of the mouse, even if it is totally hidden from view!

Examples:

```
hide
hide
show
show
show
show
hide on
```

See SHOW

SHOW *(Activate the mouse pointer)*

This redisplays the mouse hidden with the HIDE instruction. As with HIDE there's also a version of SHOW which shows the mouse, no matter how many HIDE commands have been executed. This is called using:

```
show on
```

See HIDE for more details.

Reading the joystick

STOS Basic includes six functions which make it very easy for you to detect the movements of a joystick placed in the right joystick socket.

=JOY *(Read joystick)*

d=JOY

This function returns a binary number which represents the current status of the joystick. Each of these bits are set to 1 if the test proves positive and otherwise zero. Here is a list of the various bits and their meanings:

Bit number	Significance
0	Joystick moved up
1	Joystick moved down
2	Joystick moved left
3	Joystick moved right
4	Fire button pressed

Don't worry if you are not familiar with this binary notation as you can also access each of the directions individually with the functions JLEFT, JRIGHT, JUP, JDOWN, and FIRE.

Here is a simple example to get you started.

```
load "fontset.mbk":rem From accessory disc
10 rem Move a sprite with a joystick
20 rem Set direction arrays
30 dim DX(15),DY(15)
40 S=2 : X1=160 : Y1=100
50 for I=1 to 15 : read X,Y : DX(I)=X*S : DY(I)=Y*S : next I
60 sprite 1,X1,Y1,40 : J=joy and 15 : X1=X1+DX(J) : Y1=Y1+DY(J) : if joy>15 then
X1=160 : Y1=100 : goto 60 else 60
70 data 0,-1,0,1,0,0,-1,0,-1,-1,-1,1
80 data 0,0,1,0,1,-1,1,1,0,0,0,0,0,0,0,0,0,0,0,0
```

Note that we've used the variable s to set the sensitivity of the joystick. Reasonable values range from 1(low) to 5(incredibly high).

=JLEFT *(Test joystick movement left)*

x=JLEFT

JLEFT returns a value of TRUE (-1) if the joystick has been moved left, otherwise FALSE (0). It can be used in an IF...THEN statement like this:

```
if jleft then print "LEFT"
```

=JRIGHT *(Test joystick movement right)*

x=JRIGHT

JRIGHT tests the joystick and returns TRUE (-1) if has been moved right, otherwise it returns a value of FALSE (0).

See JLEFT, JUP, JDOWN

=JUP *(Test joytsick movement up)*

x=JUP

JUP returns TRUE (-1) if joystick has been moved up, otherwise FALSE (0).

See JRIGHT, JLEFT, JDOWN

=JDOWN *(Test joystick movement down)*

x=JDOWN

The JDOWN function returns the value TRUE (-1) if the joystick has been pulled down, otherwise it returns FALSE (0).

See JRIGHT, JLEFT, JUP

=FIRE *(Test fire button state)*

x=FIRE

This function only returns a value of TRUE (-1) if the fire button on the joystick has been pressed.

See JUP, JDOWN, JLEFT, JRIGHT, JOY

Detecting collisions with a sprite

COLLIDE *(Detect collisions between two sprites)*

t=COLLIDE(n,w,h)

This provides you with an easy way of testing to see whether two or more sprites have collided on the screen. *n* refers to the sprite you wish to check and can range from 0-15, with 0 denoting the mouse pointer. *w* and *h* determine the sensitivity of the test. You can think of w and h defining the width and height of a rectangular box starting from the Hot Spot of the sprite. Whenever another sprite enters this box, a collision will be detected.

t is a number in binary format which holds a list of the sprites which have collided with sprite number *n*. Each bit in this number represents the status of the equivalent sprite. So bit 1 indicates sprite 1, bit 5 denotes sprite 5 and so on. If a collision occurs between sprite *n* and another sprite, the bit at the appropriate point is set to 1. You can test for these bits using the BTST function. If you're not technically minded, you can save yourself some trouble by adding a statement like:

```
print collide(1,10,10)
```

Place this at an important point in your program. You can now make a note of the number which is printed whenever a collision takes place. This can be tested for with a line like:

```
100 if collide(2,10,10)=6 then boom
```

Here's an example of this function in action. If you've saved the program we used in the section on ANIM, you can load this with the line:

```
load "ship.bas"
```

Otherwise you will first need to create the file zsprites.mbk in the following way:

- Load "\zoltar\zoltar.bas":rem From the games disc
- Place a fresh disc into the drive and type: save "zsprites.mbk"
- Erase the program in memory. with: new
- Load the example sprites back with load "zsprites.mbk"

You can now enter the program below:

```
5 rem Initialize screen
10 mode 0 : cls : flash off
15 rem Set colours
20 palette $0,$777,$3,$4,$17,$770,$530,$400,$655,$333,$111,
$734,$715,$706,$707,$770
25 rem Move and Animate Ship
30 sprite 2,0,40,19 : anim 2,"(19,4)(20,4)(21,4)L" : anim on 2
40 move x 2,"(1,6,80)(1,-6,80)l" : move on 2
45 rem Wait for a key press
50 wait key
55 rem Fire Missile
60 sprite 1,160,198,2 : anim 1,"(2,1)(3,1)L" : anim on
70 move y 1,"196(1,-4,60)" : move on
75 rem Test for collision
80 if collide(1,10,10)=6 then boom : goto 110
85 rem Test Missile to see if it flys off the top of the screen
90 if y sprite(1)<0 then 50
95 rem Jump Back to test
100 goto 80
105 rem Explosion
110 sprite 3,x sprite(2),40,14
120 anim 3,"(14,2)(15,2)(16,2)(17,2)(18,2)" : anim on : move off : sprite 1,-
100,100,2 : sprite 2,-100,100,9 : sprite 3,-100,100,14
```

Let's now incorporate a user-controlled ship in this scenario with the CHANGE MOUSE command.

Add the following lines to the program above:

```
21 limit mouse 0,150 to 319,198:rem Limit mouse to lower part of screen
41 change mouse 10 : rem Change mouse to picture of a ship
50 repeat : until mouse key : MX=x mouse : MY=y mouse : rem Wait for mouse
button
60 sprite 1,MX,MY+4,2 : anim 1,"(2,1)(3,1)L" : anim on
130 move off : sprite 1,-100,100,2 : sprite 2,-100,100,9
140 sprite 3,-100,100,14 : goto 30
```

This gives you a ship which can be moved around with the mouse, which can fire a missile when you press on the mouse key. You could easily detect collisions with this ship in a similar way, just by adding a line such as

```
81 if collide(0,10,10)<>1 then boom
```

Obviously you would also need to add some sort of attack capability to the defending ships as well!

You should now be in a position to understand the some of the programming techniques used in Zoltar. Although it may look rather more complicated, the

theory behind it is identical. Feel free to load Zoltar from the games disc and play around with it as much as you like.

Detecting collisions with rectangular blocks

SET ZONE *(Set a zone for testing)*

SET ZONE z,x1,y1 TO x2,y2

Defines one of 128 rectangular zones which can then be tested using the ZONE command for the presence of either the mouse or a sprite. *z* specifies a number from 1-128 which represents the zone to be created. *x1,y1* and *x2,y2* denote the coordinates of the top left and bottom right hand corners of the rectangle you wish to check.

See ZONE, RESET ZONE

=ZONE *(Tests a sprite to see if it is in a zone)*

t=ZONE(n)

This searches for the presence of sprite *n* in the list of the zones defined using SET ZONE. *n* can range from 0 to 15, with the mouse being indicated by sprite number zero as usual.

After the function has been called, *t* will hold either the number of the zone where the sprite was detected or a value of zero. Note that ZONE only returns the FIRST zone which the sprite was found. If two or more zones overlap, it is not possible to determine any other zones the sprite is also inside.

Example:

```
5 rem Muzak
6 rem Reset zones and clear screen
10 reset zone : cls back : cls physic : mode 0
15 rem Set note type
20 volume 16 : envel 9,5000
25 rem Set fill style to hollow
30 set paint 0,1,0
40 for I=0 to 7 : for J=0 to 7
45 rem Draw box
50 box I*39,J*24 to (I+1)*39,(J+1)*24 55 rem Define zones
60 set zone I*8+J+1,I*39,J*24 to (I+1)*39,(J+1)*24
70 next J : next I
75 rem Test zone and play note
80 if zone(0) then play zone(0)+20,30
90 goto 80
```

See SET ZONE, RESET ZONE

RESET ZONE *(Erase a zone)*

RESET ZONE [z]

This command erases any of the zones created by SET ZONE. If the optional *z*

is included, then only this zone will be reset. Otherwise all the zones will be deleted.

Detecting collisions with an irregular shape

=DETECT *(Find colour of pixel underneath sprite)*

c=DETECT(n)

This is a very useful command which allows you to ascertain the colour of the background pixel underneath sprite *n*. As usual, *n* can range from 0 to 15, with a value of 0 representing the mouse pointer.

After the function has executed, *c* is returned containing the colour of the point on the background screen underneath the Hot Spot of the sprite. By bordering an object with a specific colour, and then testing for this with DETECT, you can easily spot any collisions between an irregular area and the sprite.

Here is a simple example of this process.

```
load "zsprites.mbk":rem See COLLIDE for full details of how to create this

10 rem Detect demo
20 cls physic : cls back : set line $FFFF,6,0,0
30 ink 2 : arc 160,198,150,0,1800 : ink 0
40 sprite 1,rnd(314)+2,0,2 : wait vbl
50 move y 1,"(1,4,1)L" : move on
60 C=detect(1)
65 if C=2 then wait vbl : XS=x sprite(1) : YS=y sprite(1) : box XS,YS-6 to
XS+2,YS-2 : boom : goto 40
70 if y sprite(1)<200 then 60 else 40
```

Another possible application would be to detect the collision of a laser beam with a sprite. This beam could be easily created using the normal DRAW or POLYLINE commands.

Exceeding the 15 sprite limit

If you've ever seen games like Galaxians or Space Invaders you will probably consider the 15 sprite limit to be pretty restrictive. Fortunately, although you are confined to 15 moving sprites, it's easy enough to produce the illusion of dozens of actual sprites on the screen.

You can do this with judicious use of a pair of STOS Basic commands called PUT SPRITE and GET SPRITE. These allow you to create a number of copies of a sprite at once, and then just grab the ones you wish to actually move around, as and when you need them. You can add animation to these fake sprites using the SCREEN COPY and SCREEN SWAP instructions.

PUT SPRITE *(Put a copy of a sprite on the screen)*

PUT SPRITE n

Simply places a copy of sprite number *n* at its current position on the screen. Note

that the sprite you have copied is completely unaffected by this instruction.

Here is an example of how this works in practice: Load the sprites in the file ZSPRITES.MBK (See COLLIDE for details)

```
load "zsprites.mbk"
```

Now type in the following small program:

```
10 palette $0,$777,$3,$4,$17,$770,$530,$400,$555,$333,$111,
$734,$715,$706,$707,$770
20 I=8 : mode 0 : cls : flash off : hide
30 wait vbl : sprite 1,0,I,22 : rem Draw ship on the screen
40 move x 1,"0(1,8,0)e320" : move on : wait vbl
50 X=x sprite(1) : if X mod 16=8 then put sprite 1 : wait vbl
60 if X=320 then I=I+16 else 50
70 if I<192 then 30 else 90
80 goto 50
90 limit mouse : sprite 1,-100,0,22 : wait key
```

This fills the screen with dozens of copies of a single spaceship. You can now turn these ships back into movable sprites a few at a time, using GET SPRITE.

See WAIT VBL, MOVE

GET SPRITE *(Load a section of the screen into the sprite bank)*

GET SPRITE x,y,i [,mask]

This instruction enables you to grab any images off the screen and turn them into sprites. The parameters *x* and *y* refer to the start of the rectangular area to be captured.

i denotes the number of the image to be loaded, and MUST refer to an image which already exists in the sprite bank. The size of the new image is taken from the original dimensions you specified using the sprite editor. Also note that the Hot Spot of the sprite is automatically set to the point *x,y*. WARNING! This command will only work if the rectangle you are attempting to grab is completely inside the borders of the screen.

The optional *mask* specifies which colour in the new sprite is to be treated as transparent. If this mask is omitted, it will be set to zero. By changing the *mask* to a different colour you can generate a number of interesting effects. This is because the mask colour is effectively ORed with the background. A mask of zero will therefore simply display the area underneath the sprite in the normal way. Otherwise the OR operation will invariably change the colour of any of the background which shows through the sprite.

Incidentally, the *mask* has a rather different action in monochrome mode. All monochrome sprites are given a special border on the screen. The thickness of this outline is usually set to a width of one pixel, but you can increase it by including a higher value as part of the mask.

Examples:

Place the accessory disc in the drive and type:

```
load "sprdemo.mbk"
```

Now enter the following small program:

```
10 Rem Big Mouse
20 repeat:until mouse key
30 hide:
40 get sprite X mouse,Y mouse,2: change mouse 8:show
```

This borrows one of the images in the SPRDEMO file and loads it with the section of the screen underneath the mouse. It then assigns this sprite to the mouse.

We'll now look at a slightly more interesting example involving some sprites which have been placed on the screen with PUT SPRITE.

Load the file ZSPRITES.MBK from your disc. (See COLLIDE for details of how this data can be created)

```
load "zsprites.mbk"
```

Then enter the program:

```
10 rem Set colours
20 palette $0,$777,$3,$4,$17,$770,$530,$400,$555,$333,$111,
$734,$715,$706,$707,$770
25 rem Define Array P
30 dim P(20)
35 rem Reset Screen
40 hide : off : cls physic : cls back : ink 0
50 rem Copy 20 sprites on the screen
60 sprite 1,8,10,22 : rem Draw ship on the screen
70 move x 1,"8(1,4,0)e320" : move on
80 X=x sprite(1) : if X mod 16=4 then put sprite 1 : wait vbl
90 if X=320 then move off : goto 100 else 80
100 sprite 1,400,10,23 : wait key
105 rem Choose a sprite which hasn't moved
110 S=rnd(18)+1 : if P(S)=1 then 110 else P(S)=1
120 rem Get sprite
130 get sprite S*16+4,10,21
135 rem Move sprite down
140 sprite 1,S*16+4,10,21 : move y 1,"(1,4,50)" : move on
145 rem Erase sprites
150 bar S*16-4,2 to S*16+12,18
155 rem Test if sprite still falling
160 if movon(1)=0 then 110 else 160
```

This program places 20 copies of a spaceship on the screen and then animates each one in turn in an apparent violation of the 16 sprite limit. With a little more work you could easily expand the above technique to move up to 15 sprites at a time.

Sprite priority

PRIORITY ON/OFF *(Change between priority modes)*

The priority of a sprite determines how sprites are displayed when they overlap on the screen. Sprites with the higher priority always appear to have been placed in front of sprites with a lower one. Normally, the priority of the sprites is assumed

to be in REVERSE order to the sprite numbers.

You should always remember this fact when assigning numbers to your sprites. The mouse is effectively sprite number zero and therefore has the highest priority of all. This explains why the mouse always passes in front of any other sprites on the screen.

There is however, also a different priority system which can be activated with the PRIORITY ON command. This gives the highest priority to the sprites with the largest Y coordinate. So a sprite at 100 would pass above a sprite at 99 and behind a sprite at 101. In practice this option allows you to create an useful illusion of perspective. Look at the example below.

```
load "zsprites.mbk":rem See COLLIDE for details
1 rem Test of priority
5 mode 0 : cls : flash off : hide
10 priority off:rem Set normal mode
20 sprite 1,160,100,22 : sprite 2,100,94,2
30 sprite 3,100,108,19
40 move x 2,"0(1,2,160)L" : move x 3,"320(1,-2,160)L" : move on
50 wait key
60 priority on:rem Set Y mode
```

In the normal mode both of the moving sprites pass below the ship in the centre. When you select the Y priority with PRIORITY ON, the sprites are now ranked in order of their increasing Y coordinates. So sprite 3 moves above sprite 1 and sprite 2 passes behind it.

Note that if you want to create the most effective results, it's usually best to position the Hot Spot of the sprite at its base. This is because the Y coordinates used by this command relate to the position of the Hot Spot on the screen. Also notice that the PRIORITY OFF instruction can be utilised to reset the priority back to normal.

The background

Whenever a sprite is moved across the screen, it obscures some sections of the graphics and reveals others. In order to use this technique, it requires a copy of the area underneath the sprite to be held somewhere in the ST's memory. Rather than allocating a separate chunk of memory for each sprite, STOS Basic keeps a copy of the entire screen to serve as a background for the sprites.

One important consequence of this approach is that the background screen and the normal screen must always contain exactly the same image. If they don't, the sprite will tend to corrupt the area of the screen underneath when it it is moved. Therefore all STOS Basics graphics commands usually operate on both screens simultaneously. You can change this state of affairs at any time using a special AUTOBACK command.

AUTOBACK ON/OFF *(Set screen for graphics operation)*

The AUTOBACK command toggles between two different drawing modes. As a default, all graphics are sent to both the sprite background and the physical screen. The autoback feature can be turned off using the AUTO BACK OFF instruction, which leads to a substantial speed improvement in most of the graphics commands. Similarly the original mode can be reactivated with a call to AUTO BACK ON.

Example:

```
cls
```

```
autoback on:rem Set automatic background
circle 100,100,100:rem Draws a filled circle on both screens
```

Now move the mouse around on the circle. As you can see, the circle remains unchanged.

Let's try drawing the circle with AUTOBACK turned off.

```
cls
autoback off
circle 100,100,100:rem Draws a filled circle only on PHYSICAL screen.
```

If you now move the mouse on the circle, the circle will be steadily erased. This is because the sections underneath the mouse are being copied from a background screen in which the circle does not exist. By choosing the contents of the background and physical screen carefully, you can produce a number of interesting effects.

Furthermore, if your program doesn't use either the mouse pointer or the sprites, you can speed up all the graphics operations a great deal by just switching off the autoback feature using AUTO BACK OFF.

See BACK, PHYSIC, LOGIC

Miscellaneous sprite commands

UPDATE *(Change automatic sprite updates)*

Usually any sprites you draw on the screen will be automatically redisplayed whenever they are animated or moved. This feature can be temporarily halted using the UPDATE OFF command. When the updates are not active, the SPRITE, MOVE and ANIM commands apparently have no effect. In reality, they are still being operated on by the sprite instructions, but the results are simply not being displayed on the screen. You can force any sprites which have moved to be redrawn at their current positions using the UPDATE command like this:

```
update
```

Here is a summary of the three different forms of the UPDATE instruction:

UPDATE OFF	Turns off the automatic updating of the sprites. Any movements or animations appear to be suspended.
UPDATE	Redraws any sprites which have changed at their new positions. This command can occassionally be substituted for the normal WAIT VBL after a PUT SPRITE instruction, as it is much faster.
UPDATE ON	Returns the sprite updating to normal.

For an example, place the accessory disc in the drive and type:

```
new
load "sprdemo.mbk":rem Load some sprites
sprite 1,100,100,1:rem Install sprite at 100,100
```

```
move x 1"(1,1,100)(1,-1,100)l":rem Move the sprite to and fro
move on

update off:rem Stop updates
```

Remember that whilst the sprite in not being updated, it is still moving. We can demonstrate this by updating the position with:

```
update
```

To see how the sprite is progressing across the screen, type in this instruction several more times.

We can now return the sprite movements to normal with:

```
update
```

REDRAW *(Redraw the sprites)*

Redraws all the sprites at their current positions on the screen. Unlike UPDATE it takes no account of whether the sprite has been changed since the last update.

OFF *(Turn off sprites)*

This turns off all the sprite movements and animations, and removes the sprites from the screen. It is often used to reset the editor after you have broken out of a program with Control+C. As a default it is assigned to function key f10.

FREEZE *(Pause sprite and music operations)*

Temporarily halts the actions of all the sprite commands and stops any music which is currently being played. To restart these activities again simply type in the line:

```
unfreeze
```

UNFREEZE *(Restart sprite and music operations)*

Resumes any sprite movements and music halted by FREEZE.

5 Music and Sound

The Atari ST has a special sound generator which allows you to create a wide range of different effects. STOS Basic gives you complete control over this feature, and includes a variety of instructions to produce anything from a simple beep to a complex sequence of music.

Voices and tones

The ST's sound chip can play up to three notes simultaneously each performed on a separate Voice. By combining these voices, you can generate attractive harmonics. The most fundamental of the STOS Basic sound commands is PLAY.

PLAY *(Play a note)*

PLAY [voice,]pitch,duration

Plays a pure note through the loudspeaker of your TV or monitor. *Pitch* sets the tone of this sound, ranging from 0(low) to 96(high). Rather than just being an arbitrary number, each of these pitches is associated with one of the notes (A,B,C,D,E,F,G). See the following table for more details. If you specify a value of zero for the *pitch*, the note will not be produced, and PLAY will simply wait for a time specified by the *duration*.

	Octave							
	0	1	2	3	4	5	6	7
Note	Pitch							
C	1	13	25	37	49	61	73	85
C#	2	14	26	38	50	62	74	86
D	3	15	27	39	51	63	75	87
D#	4	16	28	40	52	64	76	88
E	5	17	29	41	53	65	77	89
F	6	18	30	42	54	66	78	90
F#	7	19	31	43	55	67	79	91
G	8	20	32	44	56	68	80	92
G#	9	21	33	45	57	69	81	93
A	10	22	34	46	58	70	82	94
A#	11	23	35	47	59	71	83	95
B	12	24	36	48	60	72	84	96

Duration holds the length of time the note is to be played in 50ths of a second. A duration of zero indicates that the sound will not be generated.

The optional *voice* designates which of the three voices the note is to be played on. *Voice* can range from 1-3. If it is not included then the note will be sounded on all three voices at once.

As you can see the notes go up in a cycle of 12. This cycle is known as an octave.

Here are a couple of simple examples of this function in action.

```
new
10 rem Random Music on a single voice
```

```
20 click off:rem Turn off keyboard click
30 T=rnd(96) : P=rnd(32) : play T,P : goto 30

new
10 rem Random Music on all three voices
20 click off:rem Turn off Keyboard click
30 volume 1,14 : volume 2,14 : volume 3,14
40 V=rnd(2)+1 : T=rnd(96) : P=rnd(40) : play V,T,P : goto 40

new
10 rem Example of Play
20 rem Define note arrays
30 dim A(7),A#(7),B(7),C(7),C#(7)
40 dim D(7),D#(7),E(7),F(7),F#(7)
50 dim G(7),G#(7)
60 for I=0 to 7
70 P=I*12 : C(I)=P+1 : C#(I)=P+2 : D(I)=P+3 : D#(I)=P+4
80 E(I)=P+5 : F(I)=P+6 : F#(I)=P+7 : G(I)=P+8 : G#(I)=P+9
90 A(I)=P+10 : A#(I)=P+11 : B(I)=P+12
100 next I
110 rem Define time variables
120 WN=32 : HN=16 : QN=8 : EN=4 : SN=2 : TN=1
130 rem Turn off key click
140 click off
150 rem Set volume
160 volume 15
170 rem Read note
180 read N,T : if N<0 then 230
190 rem Play note
200 play N,T
210 goto 180
220 rem Turn off sound
230 volume 0
240 click off
250 end
260 rem Music
270 data D(3),WN,E(3),WN,C(3),WN,C(2),WN,G(2),WN,-1-1
```

See CLICK OFF and VOLUME.

VOLUME *(Change the sound volume)*

VOLUME [v,]intensity

Allows you to change the volume of any subsequently generated sounds.

Intensity refers to the loudness of this sound. It can normally range from 0(silent) to 15(very loud). There's also a special setting of 16 for the envelope generator. See the ENVEL command for more details.

v indicates which of the three voices is to be regulated by the command. This number can take any value from 1 to 3. As with PLAY, if no voice is specified then all three voices are affected.

Examples:

```
click off
```

```
volume 15
play 40,10
volume 5
play 40,10

new
10 for i=0 to 15
20 volume i
30 print "VOLUME";i
40 play 60,10
50 next i
```

See ENVEL, PLAY

CLICK OFF/ON *(Turn off keyboard click)*

One minor problem you may encounter when using PLAY, is that the keyboard beeps tend to interfere with the note. Try typing the following line:

```
volume 10: play 40,1000:rem Generate a tone 20 seconds long
```

If you now hit one of the keys while the note is playing, the note will immediately stop. Since this could be very inconvenient, STOS Basic allows you to turn off the keyboard click at any time with the instruction:

```
click off
```

As you might expect, the click can be reactivated by CLICK ON. Incidentally, it is important to note that this problem does not occur when using music created by the MUSIC accessory.

The MUSIC command

Although the PLAY command is very useful for the generation of single tones, it's not really suitable for the creation of real music. The most serious problem with PLAY is that it delays the entire program for the duration of the note. What is really required is an instruction which would play a piece of music while a program was doing something else. This would allow you to add a soundtrack to a game, without spoiling any of the action. Fortunately, STOS Basic incorporates a powerful series of commands which enable you to do precisely that.

MUSIC *(Play a piece of music using interrupts)*

Plays some music which has been previously composed using the MUSIC.ACB accessory. This music is always placed by the system into bank number three.

There are four different forms of the MUSIC statement.

MUSIC N (Play tune number *n*)

The standard MUSIC instruction plays a tune in bank 3, specified by the number *n*. Note that unlike PLAY, the music is played automatically by the system, without slowing down your program in the slightest. *n* can range from 1 to the number of tunes which are currently installed (up to a maximum of 32). Here's a small example to demonstrate this process.

First load a melody from the accessory disc with the line:

```
load "music.mbk"
```

You can play this with the MUSIC instruction like so:

```
music 1
```

This music will now play in the background independently of the rest of STOS Basic. You can run, list, or even load a program without interfering with it in any way. The MUSIC command can therefore be used to add an attractive soundtrack to any of your programs. Examples of this technique can be found in the games Zoltar and Bullet Train.

MUSIC OFF (Turn off music)

The MUSIC OFF command stops a piece of music which is currently being played. You can restart this music from the beginning with MENU ON.

MUSIC FREEZE (Temporarily stop a piece of music)

Unlike MUSIC OFF, this instruction only halts the music temporarily. If it is re-entered using MUSIC ON, the music is continued from the point it was frozen. The most common use of MUSIC FREEZE is to stop a piece of music before you generate another sound effect such as an explosion. (See BANG, SHOOT, BELL, NOISE, ENVELOPE)

MUSIC ON (Restart a piece of music)

MUSIC ON resumes the current music halted by either the MUSIC OFF or the MUSIC FREEZE commands.

Example:

```
load "music.mbk":rem If it has already been loaded, omit this step
music 1:rem Play music
music off
music on:rem Restart music from the begining
music freeze
music on
```

See TEMPO, TRANSPOSE, ENVEL

TEMPO *(Change the speed of a sample of music)*

TEMPO s

Allows you to modify the speed of any tune played with the MUSIC command. *s* is the new speed, and can range from 1 (very slow) to 100 (very fast).

Place the accessory disc in the current drive and type:

```
new
load "musdemo.mb"k:rem Load music
music 1:rem Play music
tempo 100:rem Set music playing very fast
tempo 10:rem Start music playing very slow
```

See MUSIC, TRANSPOSE.

TRANSPOSE *(Change the pitch of a piece of music)*

TRANSPOSE df

Alters the pitch of a piece of music by adding the value of *df* to each note before it is played. *df* can range from -90 to +90. Negative numbers lower the note and positive numbers increase it. A *df* increment of 1, by the way, corresponds to a single semi-tone.

Load the music demo with the lines:

```
load "music.mbk"
```

Now play the music and use TRANSPOSE:

```
music 1
transpose 1:rem Increase the pitch by one semi-tone
transpose 10:rem Increase pitch by 10 semi-tones
transpose -20:rem Lower the pitch by 20 semi-tones
```

See MUSIC, TEMPO

PVOICE *(Return position in music)*

p=PVOICE(v)

PVOICE is a special command which allows you to find your position in some music you are playing. *v* refers to the voice you wish to test, and p to the position. It is important to understand that *p* is set to a number representing the address of the note and **not** to the note itself. If a number of zero is returned by PVOICE, then no music is being played on voice *v*. The PVOICE instruction enables you to determine when the music reaches a particular point and stop it if required.

Example:

Put the accessory disc into the drive and type:

```
new
10 load "music.mbk"
20 music 2
30 tempo 5
40 home : print pvoice(1),pvoice(2),pvoice(3)
50 if inkey$="" then 40
60 music off
```

This displays a number denoting the note which is being currently played. See how we used the TEMPO command to slow things down.

You can now amend the program to stop the music at a specific stage like this:

```
30 tempo 40
45 if pvoice(1)=118 then 60
```

If you run this program, the music is halted when PVOICE(1) reaches position 118.

VOICE *(Turn on/off a voice)*

VOICE OFF [v]

Lets you turn off one or more voices of a tune played by MUSIC. The optional voice *v* can take the numbers from 1-3 and specifies that only a single component of the music will be suspended. If it is not included then all three voices will be deactivated.

VOICE ON [v]

Restarts some music halted by the VOICE ON instruction. As before, *v* indicates which of the three voices is to be set in motion. If it is not specified then all three voices are set in motion.

Examples:

Place the accessory disc into the drive and type:

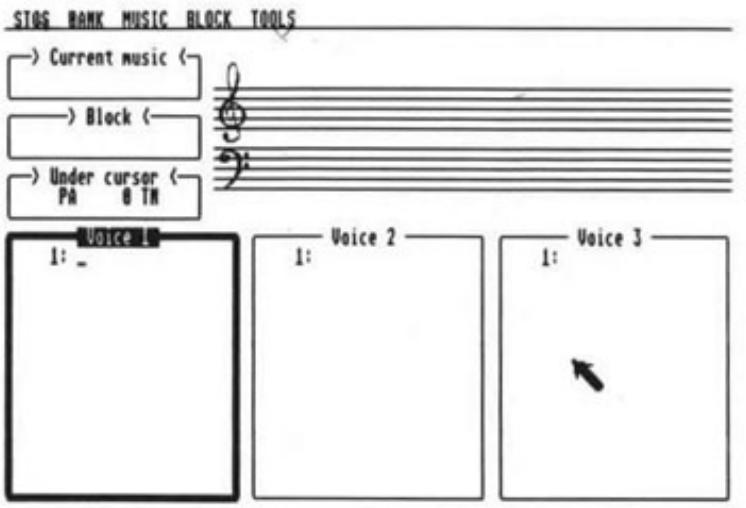
```
new
load "music.mbk
music 1
voice off 1
voice off 2
voice off 3
voice on 2
voice on 1
voice on 3
```

The Music definer

STOS Basic includes a powerful accessory in the file MUSIC.ACB. This can be used to compose a piece of music to be subsequently played with the MUSIC commands. As this is a rather large program, users of the 520ST should always remove all other accessories from memory before loading.

```
accnew:accload "music.acb"
```

You can now enter the accessory by pressing HELP+F1.



This screen consists of three windows which correspond to the three voices. Each of these windows can hold a separate component of the music. You can move between the windows using either the mouse, or the left and right cursor keys.

Above these windows is a set of menus and a graphical display of the current tune in a standard musical notation. Don't worry if you can't read music, as this window is only there as a convenient aid for those who can. The following diagram should make the format of the main screen a little clearer.

Musical notes can be entered in any of the three windows just by moving the cursor to the appropriate point and typing them directly from the keyboard. These notes are split into three distinct parts. The first section consists of the name of the note, which is input using standard musical notation, and can be one of the following 12 possibilities:

C,C#,D,D#,E,F,F#,G,G#,A,A#,B

We've listed these notes in order of increasing pitch.

The second part of a note is the octave, which can range from 0 (very low) to 7 (very high). The higher the octave, the higher the note.

Finally, each tone has a duration specified in units of a single note. This is set by the instructions in the table below.

Duration of note	Meaning
WN	Whole Note
HN	Half Note
QN	Quarter Note
EN	Eighth Note
SN	Sixteenth Note
TN	Thirtysecondth Note

You can also add an additional half note to each of these durations except the SN, by using the "." character. So QN. is a duration of a quarter of a note plus a half – three quarters of a note. Each of these sections are combined into a single string such as:

F #3 TN

You enter these notes by moving the cursor over the voice window using either the up and down arrow keys or the mouse, and then typing a command followed by a Return. You can also use the function keys to move the cursor as follows.

f2 Displays the next page of your music
f1 Displays the previous page
f3 Jumps to the start of the music
f4 Jumps to the end

When you require to enter rests into the stave you only have to enter a value of 0 for the note followed by its length.

The music instructions

In addition to simple notes, the Music definer also supports a range of other instructions which can be executed at any point in your music. Here is a list of the various options.

VOLUME v *(Set volume)*

Sets the volume of the current voice to *v*, where *v* can vary from 0 (silent) to 15 (very loud). If this instruction is not used, then a volume of 15 is set as a default.

ENVEL e *(Set envelope)*

Allows you to choose one of a number of different waveforms for your music. These waveforms determine the shape of the note by changing the volume over a period of time. *e* refers to the envelope number. As a default eight of these envelopes are already defined, although these can be readily changed using the built-in Envelope editor. See the section on this utility for more details. Each piece of music **must** contain one of these instructions at the beginning, or the tune will not be played.

Tremolo t *(Set tremolo)*

Identical to an envelope except that, instead of the volume being changed, it is the pitch of the note that is progressively altered. This adds a pleasant waver to the note. *t* is the number of the tremolo to be used. As with the envelopes, eight of these tremolos are automatically defined. Existing tremolos can be modified and new ones created with the Tremolo definer utility.

STOP TREMOLO

Deactivates the current tremolo if one is being used.

NOISE n *(Start noise)*

Generates a hiss of pitch *n* at the same time as the notes are being played by the current voice. The frequency of this sound ranges between 0 and 31. See the STOS Basic NOISE command for more details.

STOP NOISE *(Stop the noise effect)*

Turns off a noise created with NOISE.

NOISE ONLY *(Plug each note as noise rather than a pure tone)*

Plays each note as a noise rather than a pure tone. This can be used to create a number of interesting percussion effects.

MUSIC *(Reset to music)*

If the voice has been defined as NOISE ONLY, this returns the voice back to normal. Do **not** confuse with the MUSIC command from STOS Basic!

REPEAT n,p *(Repeat a section of music)*

Repeats the notes starting from the instruction number *p* to the end of the current voice. *n* refers to the number of times the music will be repeated. If a value of 0 is used for *n*, the music will be played indefinitely. **Warning:** This instruction must always be placed **before** the music to be repeated. If it is placed inside the loop, then the music will never end, as the repeat is reinitialised every time it is executed.

NTREMOLO t *(Set noise tremolo)*

Uses the Noise generator rather than a pure tone to create tremolo number *t*. The result is very odd indeed, but might occasionally be useful when used as part of a soundtrack.

NTREMOLO OFF *(Noise tremolo off)*

Turns the NTREMOLO function off.

Envelopes and tremolos

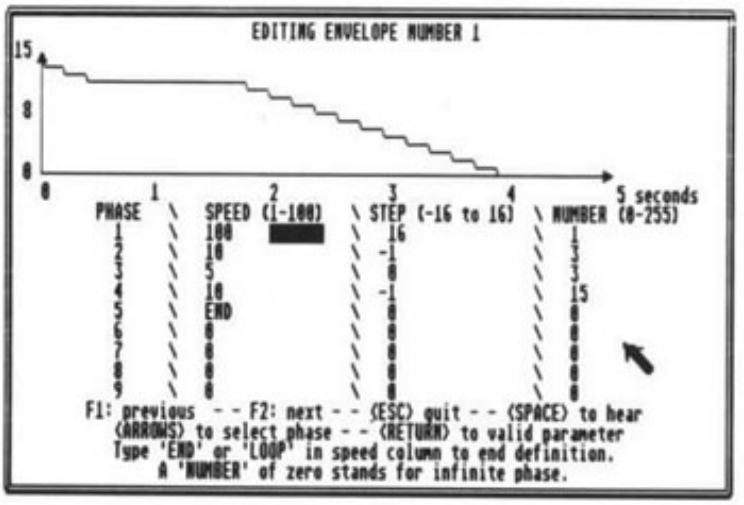
Envelopes control the evolution of the volume of a note over time. These envelopes can be created using a powerful utility built into the music definer. You can use this facility to mimic the sound of a range of different musical instruments.

Tremolos are really very similar to envelopes except that the **pitch** of the sound rather than the volume changes during the note. Tremolos can used to produce a number of interesting vibratto effects. Like envelopes they can also be edited using a special utility.

The Envelope editor

The Envelope and Tremolo editors are effectively one and the same. They can be accessed at any time using the FIX ENVELOPE or FIX TREMOLO options from the tools menu.

Since the two routines are otherwise identical, we'll concentrate on the Envelope editor. When you enter this, the following screen is displayed.



The top of the screen contains a graphical representation of the current envelope. Below this there are three windows. You can move between these using the cursor keys.

The nature of an envelope is determined by up to eight different phases. These phases are specified using the information you have entered into the windows.

The Speed window sets the speed of the phase. Possible speeds range from 1 (slow) to 100 (fast). This number indicates the delay between each step of the waveform. A speed of 100 signifies that the steps will be performed every 50th of a second, while a speed of 1 denotes an interval of 100/50 or 2 seconds between successive stages. In addition, you can also input the commands END or LOOP. END simply terminates the envelope at the current point. LOOP is rather more interesting and repeats the entire envelope, which now overlays a continuous rhythmn on any music you subsequently play.

The Step window inputs the change in the volume to be produced in each

stage. Positive numbers increase the volume, while negative numbers decrease it.

Finally there is the Number setting which determines the number of times each phase will be executed. This can range from 0 to 255.

At the start of the session you are presented with waveform number one. You can move to the next envelope by pressing f2 and to the previous one with f1.

Now for a simple example. In this we will be defining a new waveform for envelope 9. Press f2 until the number 9 is displayed at the top of the screen. Move the cursor to the first row of the Speed window and type in the following lines, terminated by Return.

```
40
30
15
```

As you can see, an END instruction is placed automatically at the end of your envelope. You should now add the steps of these phases by moving the cursor to the top of the step window and entering:

```
2
0
-1
```

Similarly you can input the number of times each stage should be performed into the Number window.

```
10
10
15
```

The envelope will now be displayed on the screen. This consists of a sharp increase in volume (attack), followed by a brief period when the volume stays the same (sustain), and a slow drop (decay). Press the spacebar to hear how this envelope actually sounds. Now move the cursor to the END statement and change it to a LOOP. This will repeat the waveform continuously.

The pull-down menus

STOS

ACKNOWLEDGMENTS	
QUIT	Exit to STOS Basic Editor.
QUIT and GRAB	Exit to STOS Basic Editor, and load the current music into bank 3.

BANK

LOAD MUSIC BANK	Load a memory bank containing a sample of music from the disc. Note that this command does **not** affect the music currently being edited. This allows you to merge two sections of music together.
SAVE MUSIC BANK	Save the music on to the disc. The name of the file **must** end with the extension .MBK.

GRAB	Grab some music from the current STOS Basic program.
ERASE MUSIC BANK	Deletes any MUSIC currently stored by the definer.

MUSIC

NEW MUSIC	Deletes the music currently being edited, and asks for the name of the new tune you wish to create. Does **not** affect any of the music held in bank 3.
RENAME MUSIC	Changes the name of the current piece of music.
PUT MUSIC	Copies the currently edited tune into one of the 32 different slots in bank 3. Bank 3 is used by STOS Basic to hold your music and is limited to a maximum of 32k. This should easily be sufficient for all practical purposes. Since the definer only saves the data which has been previously installed in the bank, you must always remember to use the PUT instruction prior to saving your music to the disc. OTHERWISE YOUR MUSIC WILL BE LOST FOREVER!
GET MUSIC	This option loads a sequence of music stored in bank 3 into the music editor. If you change this music, don't forget to place it into the memory bank with PUT, otherwise all your amendments will be lost. Incidentally, GET MUSIC automatically appends any envelopes or tremolos used by your composition into the existing set. You are, however, restricted to a maximum of 25 envelopes and tremolos at a time.
ERASE MUSIC	Allows you to delete one of the sections of music from the bank.
PLAY MUSIC	Enables you to play a piece of music you have stored in the memory bank. If you wish to play the music you are currently editing, you need to load it into the bank first using PUT MUSIC.
PUT and PLAY	Permits you to put the current music into bank 3 and then play it using just one operation.
PRINT MUSIC	Outputs a listing of the music you are editing to a printer. All three voices are printed out.

BLOCK

START BLOCK	Sets the start of a block at the current cursor position. All text below this line is subsequently displayed in inverse.
END BLOCK	Sets the end of the block. The section of the music making up this block is inverted. This block can now be manipulated with COPY BLOCK and TRANSPOSE BLOCK.

CANCEL BLOCK	Aborts current block and redisplays the section of music in normal type.
COPY BLOCK	Places a copy of the currently defined block at the cursor position. This feature can be used to copy music from one voice to another.
ERASE BLOCK	Erases the part of the music selected using the START and END BLOCK commands.
TRANSPOSE BLOCK	Allows you to add or subtract a specific number of semitones from the music in the current block. The editor expects you to input a number from -90 to +90. As with TRANSPOSE from Basic, negative values lower the pitch and positive values increase it.

TOOLS

FIX ENVELOPE	Enter ENVELOPE Editor.
FIX TREMOLO	Edit Tremolos.
ERASE ENV/TREM	Delete all envelopes and tremolos from memory.

Creating a piece of music

In order to create some music, first enter the Music Definer using Help+f1. Now move the cursor to the first voice and type:

```
ENVEL 1
```

As you press Return, you will be prompted for an eight character name for your music. In this example you can call the music anything you like. The ENVEL instruction sets the waveform of the notes which will be played. Up to 16 of these waveforms are available at any time, and these can be defined using a built-in envelope editor. Each piece of music needs to have its own envelope setting. If you omit this instruction the music will not be produced.

Move the cursor to the line below the ENVEL command and type:

```
D 3 WN
E 3 WN
C 3 WN
C 2 WN
G 2 WN
```

When you enter each line the cursor moves down one place, and the appropriate note appears on the screen. The Insert key inserts a space at the current cursor position and moves the rest of the music down a line. Similarly the Delete key can be used to erase the note under the cursor.

You can now register your music into the memory bank using the PUT option from the Music menu. This puts the tune into one of 32 different slots. These slots have numbers ranging from 1-32 and refer to the numbers used by any subsequent MUSIC command in your program. Move the mouse to slot number 1 and press the left button to install your music into the bank.

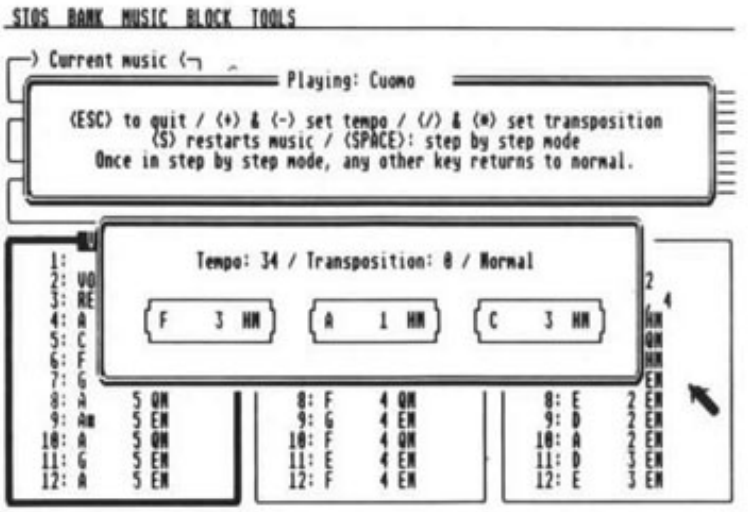
In order to listen to this music, you must select the PLAY option. As before you

need to choose the name of the music with the mouse.

Press the S key to play the music. If you're a science fiction fan, you may recognise it as part of the theme from Close Encounters of the Third Kind.

The speed of this piece can be changed while the music is playing by hitting the + and - keys, and you can alter the pitch with / and *. While the music plays, each note is displayed on the screen.

After you have finished listening to the music, you can exit back to the main menu by pressing the Escape key.



B

One minor problem with this tune was that it stopped playing after the last note. STOS Basic includes a useful REPEAT instruction which can be used in this situation. Move the cursor to the line containing the first note, and press Insert. Now enter the instruction:

REPEAT 0,3

S

The REPEAT command takes two parameters. The first number specifies how many times the music should be repeated. A value of zero indicates that the music should be played continuously. The second number holds the starting position of the notes to be repeated. This figure includes any instructions such as REPEAT or ENVEL.

E

Now go to the Music menu and choose the Put and Play option, which combines the actions of the separate Put and Play menus into a single operation. When you play the tune, it will be repeated when it reaches the end.

Try adding each of the following instructions into the music in turn. Place them just after the REPEAT command, and test the effect with Put and Play.

NOISE ONLY

B

Produces a literally off-beat effect.

ENVEL 5

Plays the five tones using envelope number 5.

TREMOLO 2

Adds a nice waver to the tone.

You can then save the music using the Save Music option from the Bank menu, or incorporate it directly into your current program with the QUIT and GRAB option. If you select the latter option you will be returned to the Basic Editor, and your

music will be automatically loaded into bank 3. You can now play this sequence by typing:

```
music 1
```

When you've heard enough, turn the music off with:

```
music off
```

We'll now provide you with another example which demonstrates how several different voices can be combined to produce a pleasent harmonic effect. Enter the Music definer with Help+f1 as before.

Move the cursor to the voice 1 window and enter the following. You don't actually have to type every entry as the last instruction is entered automatically if you press Return.

```
VOLUME 15
ENVEL 1
C 4 QN
C 4 QN
C 4 QN
D 4 QN
E 4 HN
D 4 HN
C 4 QN
E 4 QN
D 4 QN
D 4 QN
C 4 WN
```

Now move the cursor to the second window with the Right arrow and enter the next voice.

```
VOLUME 12
ENVEL 2
C 3 QN
G 3 QN
E 3 QN
G 3 QN
C 3 QN
G 3 QN
F 3 QN
G 3 QN
D 3 QN
G 3 QN
F 3 QN
G 3 QN
C 3 WN
```

You can now play this music using the Put and Play option.

Finally, we'll have a brief look at the Music example on the accessory disc. Place this disc into your current drive, and load the file MUSIC.MBK using the Load Music option.

If you call up the PLAY command, you will find that a piece of music has been loaded into slot 1 with the name Cuomo. Access this by selecting the music with the mouse. As usual you can change the tempo and the pitch of the music with the +- and */ keys respectively.

We'll now show you how you can modify the music. Jump back to the main screen with Escape and load the music into the editor with the Get Music option. Now move the cursor to the start of the first voice and hit the Insert key.

A space will be inserted into the music, and you should type in the following command:

```
TREMOLO 2
```

Select the Put and PLAY option and place the new music into the second slot. This music will be played using tremolo number 2. The difference should be obvious!

Predefined sound effects

In addition to the music commands detailed above, STOS Basic also provides you with a number of instructions which allow you to generate special sound effects for your games.

BOOM *(Generate a noise sounding like an explosion)*

As the keyboard click interferes with this sound, it's a good idea to turn it off with CLICK OFF. You should also halt any music which is currently being played, because this will be distorted by the boom. Use the command MUSIC FREEZE for this purpose.

Example:

```
new
10 click off
20 boom
30 print "You're DEAD!"
40 click on
```

SHOOT *(Create a noise like a gun firing)*

SHOOT simply produces a sound of a shot being fired.

Example:

```
new
10 click off
20 shoot
30 print "You're DEAD!"
40 click off
```

BELL *(Simple bell sound)*

Example:

```
bell
```

Defining your own effects

So far we've only looked at the pre-defined effects, but you can also use the NOISE command and the ENVELOPE instruction to generate a vast range of other useful sounds.

NOISE

NOISE v,p

NOISE produces a sound like a rushing wind. The frequency of this noise is set by the pitch *p*, where *p* is a number from 1 (very high) to 31 (very low). *v* specifies the voice which the noise is to be played on. If it is not included the noise is output to all three voices simultaneously. Note that any noise generated with this command can be played continually while a program is running – just like the MUSIC command.

Example:

```
new
10 click off
20 for i=1 to 32
30 noise I
40 wait key
50 next i
```

The NOISE command really comes into its own when used in conjuction with the ENVEL instruction.

ENVEL

ENVEL type,speed

ENVEL activates one of the ST's 16 different envelopes. These periodically alter the volume of a sound created with either NOISE or PLAY. *type* specifies the type of envelope to be used and can take any value from 1 to 15. *speed* ranges from 1 (very fast) to 66535 (very slow) and determines the length of the sound. Before you can use this feature, you must first set the volume to 16 with VOLUME.

Example:

```
volume 16:rem Set volume
noise 10:rem Create a noise of pitch 10
envel 10,100:rem Shape the sound using envelope 10
envel 10,1000:rem Helicopter sound
```

As you can see, it is possible to utilise ENVEL to produce a number of interesting effects.

Here is a small program to help you to explore the various possibilities of this instruction.

```
10 rem Program to experiment with the NOISE
20 rem and the ENVEL instructions
30 cls 35 locate 0,0 : input "Input length of the sound from 1-10000";T
40 locate 0,0 : print "Press a key to scroll through the sounds  "
50 click off
60 for J=0 to 15
70 envel J,T
80 for I=1 to 31
90 noise I
100 locate 10,10 : print "Envelope";J;"  ";
110 locate 10,11 : print "pitch  ";I;"  ";
120 wait key
```

```
130 next I
140 next J
150 input "Continue Y or N";A$ 160 if A$="Y" or A$="y" then 35
```

These envelopes can also be used to shape the pure tones generated by a PLAY command.

Example:

```
click off
volume 16
envel 8 ,100
play 37,30
```

You can explore these effects using the program above by typing the following lines:

```
35 locate 0,0 : input "Input length of sound from 1-100";T
36 input "Starting envelope 1-15";S
37 if S<1 or S>15 then print "Bad Envelope number " : goto 36
60 for J=S to 15
80 for I=1 to 96 step 3
90 play I,T
```

Note that the variable *t* refers to the time the note will be played in 50ths of a second. When using the above routine, it's always a good idea to keep a pen and paper handy to write down any sounds you want to keep. You will be amazed at some of the noises which can be achieved with these commands.

As a general rule, NOISE is best suited for the creation of mechanical sounds such as engines and machine guns. PLAY can generate more unusual effects – like laser beams and alarms.

See NOISE, PLAY and VOLUME.

6 Graphics functions

Although STOS Basic isn't Gem based, it still supports a wide range of powerful graphical functions similar to those provided by the Gem VDI. These include facilities for drawing rectangles, circles and polygons. In addition, there's also a special set of commands which make it particularly easy to create programs capable of running equally well in all three resolutions. To that end STOS Basic effortlessly allows you to change between low and medium resolution at any time within your program.

Clearing the screen

CLS *(Clear the whole screen)*

This instruction clears the entire screen at high speed. It is usually used to initialise the screen at the start of a program. CLS has a number of useful extensions which enable you to erase all or part of a screen stored anywhere in the ST's memory. A full explanation of these options can be found in Chapter 7.

Colours

The ST allows you to display up to 16 colours on the screen at any one time. These colours are chosen from a possible palette of 512. As you might expect, the number of colours which are available depends on the graphics mode the ST is currently running in. Each of the 16 colours is referred to by a number called an index. Here is a list of the various alternatives.

Resolution	Mode	Maximum no of colours	Colour Indices
Low	0	16 from 512	0 to 15
Medium	1	4 from 512	0 to 3
High	2	2 from 2	0 to 1

Before you can draw something on the ST's screen you first need to specify which colour you wish to use. This colour can be set using the INK instruction.

INK *(Set colour of graphic drawing operations)*

INK index
Index is the number of the colour to be used for all subsequent drawing operations.

Note that index number 2 is slightly unusual, in that it flashes on and off several times a second. You can produce a similar effect using the FLASH instruction covered in section 6.7.

COLOUR *(Assign a colour to an index)*

There is also a special COLOUR instruction which allows you to choose which of the 512 colours is to be used for any particular index.

COLOUR index,$RGB
Index is the number of the colour to be changed.

$RGB is usually a hexadecimal expression which determines the exact shade of the new colour.

This expression consists of three digits ranging from 0 to 7, each of which sets the strength of one of the primary colours, RED (R), GREEN (G) or BLUE (B) in the final result. Here are a few examples of this notation:

Components	Hexadecimal form	Final Colour
R=0 G=0 B=0	$000	BLACK
R=7 G=0 B=0	$700	BRIGHT RED
R=7 G=7 B=0	$770	YELLOW
R=0 G=7 B=0	$070	GREEN
R=4 G=0 B=7	$407	VIOLET
R=7 G=7 B=7	$777	WHITE
R=3 G=3 B=3	$333	GREY

So if, you want to make colour number 5 yellow, you would type:

```
COLOUR 5,$770
```

When this statement is executed, any graphics displayed on the screen which already use colour number 5, will be immediately changed to the new colour (yellow).

=COLOUR *(Read the colour assignment)*

There's also a function with the same name, which takes an index number, and finds the colour value which has been assigned to it. This is used in the following manner:

c=COLOUR(index)
c is any variable and *index* is the colour number whose shade you want to determine.

You can use this function to produce a list of the current colour settings of your ST, like this:

```
new
10 mcol=16:rem set mcol to 4 in medium res
20 for I=0 to mcol-1
30 print HEX$(colour(I),3)
40 next I
```

PALETTE *(Set the current screen colours)*

The PALETTE instruction is really just a rather more powerful version of COLOUR. Instead of loading the colour values one at a time, the PALETTE command allows you to install a whole new palette of colours in a single line.

PALETTE list of colours
This list can contain anything up to the maximum number of colours available in the current graphics mode.

To see PALETTE in action, type one of the lines below:

Invert the screen in high res:

```
PALETTE $777,$000
```

Use this line for medium res:

```
PALETTE $000,$700,$746,$534
```

Use this line for low res:

```
PALETTE $000,$700,$070,$007,$770,$077,$707,$777,
$300,$030,$003,$330,$033, $303,$333,$345
```

Drawing lines

PLOT *(Plot a single point)*

The simplest of the drawing functions provided by STOS Basic, is the plot command, which sets any point on the screen to a specific colour. The format of the PLOT instruction is just:
PLOT x,y [,index]
Plots a point at the coordinates *x,y*.

If value of *index* isn't included, then PLOT will use the colour which was chosen using INK.

In order to test this function on a colour monitor type:

```
new
10 mode 0
20 plot rnd(319),rnd(199),rnd(15)
30 goto 20
```

POINT *(Get the colour of a point)*

As with COLOUR, there is also a function to perform the reverse of this.

c=POINT(x1,y1)
POINT returns the colour of the point at the coordinates *x1,y1* in the variable *c*.

DRAW *(Draw a line)*

DRAW is another very basic instruction which allows you to draw a straight line on the ST's screen. There are two forms of the DRAW statement:

DRAW x1,y1 TO x2,y2 Draws a line between the coordinates *x1,y1* and *x2,y2*

DRAW TO x3,y3 Draws a line from the last line drawn, to *x3,y3*

Example:

```
new
 5 colour 3,$707:ink 3
10 draw 0,50 to 200,50
```

```
20 draw to 100,100
30 draw to 0,50
```

It is important to note that, in order to make DRAW operate at the maximum possible speed, this instruction has been restricted to a single line type. Because of this, any attempt to alter the line style using SET LINE will have no effect whatsoever.

See also POLYLINE, INK.

BOX *(Draw a hollow rectangle on the screen)*

BOX x1,y1 TO x2,y2
x1,y1 are the coordinates of the top left hand corner of the box.
x2,y2 are the coordinates of the point diagonally opposite.

Example:

```
box 10,10 to 200,100
```

See also SET LINE, INK, and BAR

RBOX *(Draw a rounded hollow box)*

This is almost identical to BOX, except that the edges of the rectangle are rounded. As before the format is:

RBOX x1,y1 TO x2,y2
x1,y1 is the top right corner of box and *x2,y2* is the bottom left corner.

RBOX is very useful for producing Macintosh-like borders around a piece of text.

Example:

```
new
 5 colour 3,$7:ink 3
10 rbox 156,100 to 245,130
20 locate 20,10: print "testing..."
```

See SET LINE, INK and RBAR.

POLYLINE *(Multiple line drawing)*

POLYLINE is a very powerful instruction indeed as it enables you to generate complex hollow polygons using just a single line of code.

POLYLINE x1,y1 TO x2,y2 TO x3,y3 ...
Where *x1,y1* = coordinates of point 1, *x2,y2* = point 2 and *x3,y3* = point 3

POLYLINE first draws a line from point 1 to point 2, and then another line from point 2 to point 3. It then repeats this procedure, and draws a line between each successive pair of points until it reaches the end of the list. This means that POLYLINE is roughly equivalent to the lines.

```
DRAW x1,y1 TO x2,y2
DRAW TO x3,y3
```

Now type in the following line, which draws a triangle on the ST's screen:

```
polyline 0,20 to 200,20 to 100,100 to 0,20
```

Notice how I've used four pairs of coordinates to draw three lines. As a general rule, in order to create a closed polygon, the last group of coordinates should always be the same as the first.

Also see SET LINE, INK and POLYGON.

ARC *(Draw a circular arc)*

ARC draws a segment of a circle on the screen. It is specified by:

ARC x1,y1,r,startangle,endangle
x1,y1 are the coordinates of the centre of the circle, r is its radius.

Startangle is the angle the arc should be started from, and *endangle* is the angle at which it should finish.

Angles are measured in units of a tenth of a degree, and can therefore range from 0 to 3600. Think of a clockface – an angle of 0 would now correspond to the direction pointed at by the short hand at three o'clock. Also, since STOS measures all the angles in an anti-clockwise direction, an angle of 900 would be represented by a time of twelve o'clock, and the maximum angle (3599) would be at approximately 3:01.

The following program should make this a little clearer:

```
new
10 draw 100,120 to 190,120
20 for a=0 to 3600 step 10
30 arc 100,120,90,0,a
40 next a
```

Notice that this function is also able to produce a unfilled circle:

```
ARC x1,y1,r,0,3600
```

Try:

```
arc 100,100,100,0,3600
```

See SET LINE, INK, PIE and CIRCLE

EARC *(Draw an elliptical arc)*

The EARC instruction is very similar to ARC, but produces an elliptical arc rather than a circular one.

EARC x1,y1,r1,r2,startangle,endangle
x1,y1 are coordinates of the centre of the arc, *startangle* and *endangle* the angles of the start and the end of the arc *r1* and *r2* specify the size of the two radii of the ellipse.

If you're not mathematically minded, it may help to consider *r2* to be the vertical part of the radius, and *r1* the horizontal. When *r1* and *r2* are the same, the ellipse

will be almost identical to a circle. If *r2* is much greater than *r1* then the ellipse will be tall and thin, and if the reverse is true, it will be short and wide.

You can use this function to draw a complete ellipse using:

```
earc x1,y1,r1,r2,0,3600
```

Example:

```
earc 100,100,30,50,0,3600
```

Example:

```
new
10 cls:colour 1,$47:ink 1
20 draw 120,119 to 160,119
30 for R1=40 to 80 step 40
40 for R2=40 to 80 step 40
50 for A=0 to 3600 step 200
60 earc 120,119,R1,R2,0,A
70 next A
80 next R2
90 next R1
```

Line Types

So far in our examples, we have restricted ourselves to using solid lines. But STOS Basic also allows you to use a wide variety of other line styles. These can be used to great effect, in anything from the creation of simple diagrams to complex drawing routines.

SET LINE *(Set the line styles)*

SET LINE mask,thickness,startpoint,endpoint

Mask is the bitmap for the line, and *thickness* can range from 1 (very thin) to 40 (extremely wide). *Startpoint* and *endpoint* specify one of three styles to be used at the beginning and the end of every line: 0=SQUARED, 1=ARROWED, 2=ROUNDED.

Mask is a 16-bit binary number which holds a so-called bitmap of the line. In this system, any points in the line which are to be displayed in the ink colour are represented by the binary digit 1, and any points which are to be set to the background colour are represented by a zero.

So a normal line is denoted by the binary number %1111111111111111 and will be displayed as: _____
and a dotted line like: _ _ _ _ will be produced by a mask of %1111000011110000
By setting the line mask to numbers between 0 and 65535 it is possible to generate an almost infinite variety of different line types.

The program below contains a number of examples of this function in action.

```
new
10 cls: colour 3,$770 : ink 3
20 set line %1111111111111111,10,0,1
```

```
25 rem A large arrow
30 arc 100,199,90,0,1800
35 rem A dotted diagonal line
40 set line %1111000011110000,1,0,0
50 polyline 200,60 to 300,100
55 rem A single large point
60 set line %1111111111111111,20,0,0
70 polyline 100,150 to 100,160
```

Notice how we've used POLYLINE instead of DRAW and POINT. This is because neither of these instructions are capable of using the line styles installed by SET LINE.

See INK, POLYLINE, BOX, RBOX, ARC and EARC.

Filled Shapes

STOS Basic includes a number of useful instructions to enable you to create a wide range of filled shapes.

PAINT *(Contour fill)*

The PAINT command allows you to fill any existing hollow surfaces on the ST's screen with colour. As you might expect, this colour can be set with the INK instruction. In addition, you can also use SET PAINT to specify one of a number of different fill patterns.

PAINT x1,y1
x1,y1 are the coordinates of a point inside the object to be filled.

Look at the following example:

```
new
10 colour 3,$604:ink 3 ink 3
20 box 0,10 to 100,100
30 box 50,60 to 150,150
40 ink 1
50 paint 70,70
```

PAINT will happily fill any surface you like providing it is completely enclosed by lines. If however, there is a gap in one of these lines, the fill colour will leak out into the rest of the screen. The effect of this can be seen by adding line 15 to the above example:

```
15 set line %1111000011110000,1,0,0
```

Incidentally, PAINT corresponds directly to the FILL instruction found in other versions of Basic. Take care not to confuse the two as the STOS Basic FILL command has a very different effect!

BAR *(Draw a filled rectangle)*

This draws a filled bar using the current ink colour.

BAR x1,y1 TO x2,y2
x1,y1 hold the coordinates of the top left corner of the bar, *x2,y2* the coordinates of the corner diagonally opposite.

```
new
10 mode 0
20 X1=rnd(200):Y1=rnd(100):W=rnd(100):
H=rnd(80)
30 ink rnd(15)
40 bar X1,Y1 to X1+W,Y1+H
50 goto 20
```

See also RBAR, BOX, SET PAINT and INK

RBAR *(Draw a filled rounded rectangle)*

RBAR draws a filled and rounded rectangle on the screen.

RBAR x1,y1 TO x2,y2
x1,y1 hold the starting corner of the bar.
x2,y2 hold the coordinates of the corner diagonally opposite.

If you've already typed the BAR example above, you can see how this works by changing line 40 to:

```
40 rbar X1,Y1 to X1+W,Y1+H
```

Refer also to RBAR, BOX ,SET PAINT and INK

POLYGON *(Draw a filled polygon)*

The POLYGON instruction is identical to POLYLINE except for the fact that it generates a filled shape rather than a hollow one. As usual the fill colour is set using INK, and the fill pattern with SET PAINT.
POLYGON x1,y1 TO x2,y2 TO x3,y3 ...
Where *x1,y1* are the coordinates of point 1
x2,y2 those for point 2 and *x3,y3* those for point 3

Example:

```
polygon 0,20 to 200,20 to 100,100 to 0,20
```

Now type in lines 10 to 50:

```
new
10 mode 0
20 ink rnd(15)
30 X1=rnd(200):Y1=rnd(100):H=rnd(100):
W=rnd(90)
40 polygon X1,Y1 to X1+W,Y1 to X1+W/2,
Y1+H to X1,Y1
50 goto 20
```

This program fills the screen with pretty coloured triangles.

Also see POLYLINE, INK, SET PAINT.

CIRCLE *(Draw a filled circle)*

CIRCLE x1,y1,r
x1,y1 are the centre of the circle and *r* is its radius.

Example:

```
10 mode 0
20 ink rnd(15)
30 X=rnd(200):Y=rnd(100):R=rnd(90)
40 circle X,Y,R
50 goto 20
```

See ARC, INK and SET PAINT.

PIE *(Produce a pie chart)*

PIE is used to draw a segment of a circle in the current fill colour. In practice it can be considered to be a solid version of ARC. Like ARC it needs two angles, which denote the starting and the ending points of the pie chart respectively.

PIE x1,y1,r,startangle,endangle
x1,y1 are the coordinates of the centre of the chart and *r* is its radius.
Startangle and *endangle* range from 0 to 3600, where 0 is 3 o'clock, and angles increase in an anticlockwise direction.

Example:

```
10 rem Get free space on single density disc
20 rem Divide by 100 to convert into the range 0-3600 (approx)
30 rem Change to 200 for double sided drives
40 cls: colour 1,$700 : ink 1 : colour 3,$70
50 D=dfree
60 D=D/100
70 pen 3 : locate 20,2 : print "% Disk space free"
80 pen 1 : locate 20,3 : print "% Disk space used"
90 ink 3
100 pie 100,110,60,0,D
110 ink 1
120 pie 100,110,60,D,3600
```

This program displays the free space on the disc as a pie chart.

See also ARC, INK and SET PAINT.

ELLIPSE *(Draw a filled ellipse)*

The ELLIPSE instruction is used to draw a filled ellipse in much the same way that CIRCLE produces a filled circle.

ELLIPSE x1,y1,r1,r2
x1,y1 are the coordinates of the centre of the ellipse.
r1 and *r2* are the two radii.

You can now type in the following program:

```
new
10 mode 0
20 ink rnd(15)
30 X1=rnd(200):Y1=rnd(100):R1=rnd(90):R2=rnd(90)
40 ellipse X1,Y1,R1,R2
50 goto 20
```

See EARC, EPIE, INK and SET PAINT.

EPIE *(Draw an elliptical pie)*

This function corresponds directly to the EARC instruction and draws a solid elliptical pie chart.

EPIE x1,y1,r1,r2,startangle,endangle
x1,y1 are the coordinates of the centre of the segment and *r1* and *r2* its two radii. *Startangle* and *endangle* range from 0 to 3600, and rotate in an anticlockwise direction.
If the very idea of an elliptical pie chart seems ridiculous, we've included a couple of simple examples which may make you change your mind.

```
epie 100,100,100,20,0,2225
epie 110,110,100,20,2225,3600
```

As you can see, the use of ellipses lends useful impression of depth to any pie chart.

If you've already typed in the pie chart example, try adding the following lines:

```
100 epic 200,110,90,10,0,D
120 epic 200,110,90,10,D,3600
```

Fill types

STOS Basic allows you to use up to 36 different fill styles. These patterns can be grouped into four distinct types: Solid, dotted, lined, and user-defined. Furthermore, if you don't find the pattern you like, you can easily create one of your own.

SET PAINT *(Select fill pattern)*

The SET PAINT instruction has the format:

SET PAINT type, pattern, border

Type can range from 0 to 4.

The effect of the various types can be found by inspecting the table below.

Fill Type	Effect
0	Surface is not filled at all
1	Surface is filled with the current INK colour (solid)
2	Surface is filled with one of 24 dotted patterns
3	Surface is filled with one of 12 lined patterns
4	Surface is filled with a user-defined line pattern (See SET PATTERN)

The fill pattern is specified using a number, which can range between 1 and 24 or 1 and 12 depending on whether DOTTED or LINED type has been selected. If neither of these types have been chosen, pattern should be set to 1.

Border has just two possible values: 0 and 1. A border of 1 is used to indicate that the filled surface should be enclosed in a line of the current INK colour.

The following program prints out the fill types associated with each of the different styles:

```
new
10 rem Print out a list of dotted patterns
15 mode 0
20 for TYPE=2 to 3
30 if TYPE=2 then LIM=24 else LIM=12
40 for STYLE=1 to LIM
50 rem Set fill pattern with style number style and a border of 1
60 set paint TYPE,STYLE,1
70 rbar 0,0 to 310,180
80 locate 0,4:centre "Type "+str$(TYPE)+" Style " + str$(STYLE)
90 locate 0,6:centre "Press any key to continue"
100 wait key
110 next STYLE
120 next TYPE
```

Warning: Do not confuse SET PAINT with SET PATTERN!

See CIRCLE, ELLIPSE, BAR, RBAR, PIE, EPIE and POLYGON.

SET PATTERN *(Set a user-defined fill pattern)*

SET PATTERN is used to install the user-defined fill pattern specified with the instruction SET PAINT.

SET PATTERN address of pattern

Address of pattern refers to the address in the ST's memory where the new pattern is to be found.

Patterns can be stored in either a memory bank, a string or an array of integers. If you decide to store your pattern in a variable array, then you must always use the VARPTR instruction to calculate the address of this data, before you call SET PATTERN.

So if the pattern was held in the string *P$*, you would use the instruction SET PATTERN VARPTR(P$)

Each pattern is 16 points high by 16 points wide and takes up 16 two byte words of memory for each colour plane.

But how do you create this pattern in the first place? One particularly easy solution is to treat your fill pattern as just a 16 by 16 sprite. This allows you to draw any of your patterns using the sprite definer, and then load this sprite data into your program in the normal way.

LOAD "PATTERN.MBK"
(Pattern can be any set of 16x16 sprites)

Then all you need to do is work out the address of this data for use by SET PATTERN. This can be done with the following program:

```
10 rem Work out size of data
20 if mode=0 then PLANES=4
30 if mode=1 then PLANES=2
40 if mode=2 then PLANES=1
50 rem Get start of sprite information block
60 S=1 : rem Use image number 1. S can be any number up    to the current
number of sprites
70 rem Get start of sprite parameter block for image 1
90 SP=leek(start(1)+4*(mode+1))+start(1)+4
```

```
100 rem Get start of sprite parmeter block for image S
110 SPB=SP+(S-1)*8 : POS=leek(SPB)+SP+32*PLANES
120 rem Get location of sprite image
130 POS=leek(SPB)+SP+32*PLANES
140 rem Choose user-defined fill pattern
150 set paint 4,1,1
160 rem Set user pattern to image in pos
170 set pattern POS
180 rem Test new fill pattern
190 circle 100,100,100
```

If you want to know how all this actually works, please refer to the technical reference section in Chapter 12.

Special effects

FLASH *(Set flashing colour sequence)*

This command gives you the ability to periodically change the colour assigned to any colour index. It does this with an interrupt similar to that used by the sprite and the music instructions. The format of the flash instruction is:

FLASH index,"(colour, delay)(colour, delay)(colour, delay)..."

Index is the number of the colour which is to be animated.

Delay is set in units of a 50th of second.

Colour is stored in the standard RGB format (See COLOUR for more details) The action of FLASH is to take each new colour from the list in turn, and then load it into the index for a length of time specified by the delay. When the end of this list is reached, the entire sequence of colours is repeated from the start.

Note that you are only allowed to use a maximum of 16 colour changes in any one FLASH instruction. Here is a small example:

```
flash 1,"(007,10)(000,10)"
```

This alternates colour number 1 between blue and black every 10/50 (1/5th) of a second.

Now for something more complex:

```
flash 0,"(111,2)(333,2)(555,2)(777,2)(555,4)
(333,4)"
```

If this gives you a headache, you will be glad to learn that you can turn the flashing off using the instruction:

```
flash off
```

Also note that on startup, colour number 2 is a flashing colour. It's therefore a good idea to turn this off before loading any pictures from the disc.

See SHIFT and INK

SHIFT *(Colour rotation)*

SHIFT allows you to produce startling effects such as the famous Neochrome waterfall. It does this by rotating the entire palette of 512 colours into the 16 colour indeces using interrupts.

SHIFT Delay [,Start]

Delay is the delay between each rotation in 50ths of a second.
Start enables you to change only the colours with indeces greater than an initial value.

If a starting value is not included in the instruction, then the rotation will begin from colour number 1.

Here is a small example of SHIFT:

```
shift 10
```

See also FLASH, PALETTE and COLOUR.

The writing modes

Whenever you draw some graphics on the ST's screen, you normally assume that anything underneath it will be overwritten. Sometimes this can be inconvenient, and in this case it's useful to have the ability to choose a slightly different method of drawing. STOS Basic provides you with a special instruction called GR WRITING for just this purpose. The format of the statement is:

GR WRITING MODE
Where *MODE* can take the values from 1 to 4.

Replacement mode *(MODE=1)*

This is the default condition. Any existing graphics on the screen will be completely replaced by anything you draw over them.

Transparent mode *(MODE=2)*

Transparent mode informs STOS that only the parts of the drawing which are actually set to a specific colour are to be plotted. This means that any points in the new drawing which have a colour of zero, are assumed to be transparent and are therefore omitted.

XOR mode *(MODE=3)*

XOR combines your new graphics with those already on the screen, using a logical operation known as eXclusive OR. The net result of this mode is to change the colour of the areas of a drawing which overlap an existing picture. One interesting side effect of XOR mode is that you can erase any object from the screen by simply setting XOR mode and drawing your object again at exactly the same place. This technique can be used to wipe complex polygons from the screen amazingly quickly.

Example:

```
circle 100,100,100
```

```
gr writing 3
circle 100,100,100
```

Inverse transparent *(MODE=4).*

As you might expect, this mode has the opposite effect of transparent mode, and only plots points with a colour of zero. All other points in the new picture are completely ignored.

Now type in the following small example:

```
 5 mode 0
10 for I=1 to 4
20 cls
30 centre "Mode number "+str$(i)
40 gr writing I
50 set paint 1,1,1
60 bar 100,50 to 200,150
70 set paint 3,6,1
80 circle 150,100,50
90 locate 0,4:centre "Press Return to continue"
100 wait key
110 next I
```

This demonstrates the action of all four writing modes. Incidentally, the reason for the GR part of the instruction is to distinguish it from a similiar procedure called WRITING, which is used for the text operations. You should therefore take care not to confuse the two instructions.

See also AUTOBACK and WRITING

Polymarkers

What are Polymarkers?

Polymarkers are useful facilities normally provided by the Gem VDI, which enable you to plot lists of objects such as crosses, diamonds and squares as easily as a single point.

POLYMARK *(Plot a list of polymarkers)*

This instruction has the form:
POLYMARK x1,y1;x2,y2;x3,y3;.....
(x1,y1),(x2,y2),(x3,y3) are the coordinates of a list of markers to be printed on the screen.

Note that all polymarkers are drawn in the current INK colour. The marker type is assumed to be a "." by default, and can be changed using SET MARK.

Example:

```
polymark 100,100;300,120
```

This draws two markers at 100,100 and 300,120

See SET MARK and INK.

SET MARK *(Set the marker used by polymark)*

This allows you to choose the marker used by POLYMARK from a selection of six different marker types. Each polymarker can be drawn in eight sizes, ranging in 11 point increments from 6 to 83 pixels wide.

SET MARK type, size

Here is a table which illustrates the various possibilities:

Type Number	Marker Used .
1	Point "." Note this marker is only available in one size.
2	Plus sign "+"
3	Star "*"
4	Square
5	Diagonal cross
6	Diamond

Example:

```
set mark 4,83
polymark 100,100;200,100;300,100
```

This produces three squares on the screen.

Here is a much larger example which generates all the possible marker types in each of the eight sizes.

```
10 rem Displays all six polymarkers
20 rem in each of their sizes
40 mode 0
50 rem Opens a window
60 windopen 5,0,0,40,12,2,3
70 centre "POLYMARKS" : locate 0,1 : centre "Press a key"
80 rem Turn off cursor and mouse pointer
90 curs off : hide
100 for I=0 to 7
110 restore 240
120 for J=1 to 6
130 rem Change marker sizes in 11 point increments
140 set mark J,I*11+6
150 rem Get coordinates of mark
160 read X,Y
170 rem Draw a marker at X,Y
180 polymark X,Y
190 next J
200 wait key
210 next I
220 wait key
230 curs on : show
240 data 50,80,160,80,270,80
250 data 50,145,160,145,270,145
```

The square polymarkers are especially useful as they allow you to quickly create large grids on the ST's screen with just a few lines of code.

See also POLYMARK and INK.

Multi-mode graphics

In order to write programs capable of working in all three of the ST's graphics modes it's essential to be able to determine precisely which mode the ST is running in at any one time. Also, since some programs need to use a screen with the maximum possible size, it would be useful to have the ability to change between low and medium resolution when required. This feature is impossible using GEM, but in STOS Basic it's easy. To change from a low resolution screen to medium resolution you simply type:

```
mode 1
```

You are now in medium resolution. This instruction can also be placed in a STOS Basic program.

Example:

```
10 mode 1
```

MODE *(Change the graphics mode)*

MODE n
n can be either 0 or 1.

Note that since mode 2 requires a special high resolution screen, a value of 2 simply doesn't make sense. Additionally, MODE will generate an error message if you try to use it on a monochrome monitor.

There is also a MODE function which can be used to read the current graphics mode at any time.

Example:

```
10 if mode=2 then stop:rem This program will not work in high resolution
20 if mode=0 then mode=1: rem Enter medium resolution
30 centre "Medium Resolution"
40 locate 0,4:centre "Press a key"
50 wait key
60 locate 0,4:centre "Press a key"
70 centre "Low resolution"
80 wait key
```

DIVX and DIVY

Supposing you want to write a single program capable of working in all three resolutions. There are two problems you will encounter in this situation: The different number of available colours and the incompatible screen sizes. It's easy enough to solve the first difficulty just by limiting the number of colours to 2. But how do you beat the second problem? STOS Basic provides you with an answer in the variables DIVX and DIVY which hold two numbers denoting the current width and height of the display area, expressed as a fraction of those used in mono mode. Here is a small table showing the values these variables will take in all three graphics mode.

MODE	Resolution	DIVX	DIVY
0	Low	2	2
1	Medium	1	2
2	High	1	1

To draw graphics which look equally good in any resolution, all you now need to do is to assume the screen is 640 by 400, and divide all your X coordinates by DIVX and your Y coordinates by DIVY.

Type the following line:

```
rbox 0,0 to 639/divx,399/divy
```

This fills the screen with a rounded box whatever graphics mode your ST is running under.

Now for a rather larger example:

```
1 rem Simple graphics demo
10 cls
20 COLS=15: rem Assume low res at the start
30 rem Now test for medium res
40 if mode=1 then COLS=3
50 rem And for high res
55 if mode=2 then COLS=1
60  X1=rnd(319):Y1=rnd(199):W=rnd(319):H=rnd(199):C=rnd(cols):TYPE=rnd(2)
70 ink C
80 if TYPE=1 then X2=X1+W:Y2=Y1+H:box X1/divx,Y1/divy to X2/divx, Y2/divy
90 if TYPE=2 then X2=X1+W:Y2=Y1+H:rbox X1/divx,Y1/divy to X2/divx,Y2/divy
100 goto 60
```

CLIP *(Restrict all graphics to part of the screen)*

The CLIP instruction is used to restrict the actions of all the graphics commands to a rectangular region of the screen. If you attempt to draw anything outside this area, your object will be clipped to fit in this rectangle.

CLIP x1,y1 TO x2,y2
x1,y1 are the top left hand corner of the rectangle and *x2,y2* are the coordinates of the corner diagonally opposite this point.

Example:

```
new
10 cls
20 clip 50,50 to 150,150
30 box 50,50 to 150,150
40 circle 100,100,100
```

As you can see, any parts of the circle outside the clipping rectangle haven't been drawn.

This instruction is often used in conjunction with the STOS windows.

In order to turn the clipping off, simply type:

```
CLIP OFF
```

7 The screen

STOS Basic includes a powerful set of instructions which allow you to effortlessly manipulate the size and shape of the ST's screen. These commands can be utilised to produce some quite stunning effects. In this chapter we will be examining the various techniques which make this possible.

Multiple screens

STOS Basic holds two screens in memory at any one time. The first is called the Physical screen, and is the screen which is actually displayed on your televison set. There is however, also a separate Background screen which is used by the sprite commands. Normally the only difference between the two screens are the sprites, which are only drawn on the physical screen. STOS Basic uses this background to redraw any areas of the screen which are revealed underneath the sprites when they are moved. See AUTOBACK for more details.

BACK *(Address of the background screen)*

This variable holds the location of the screen used as the sprite background.

Example:

```
print back:rem Address of background is 983040 for 1040ST users

458752
```

PHYSIC *(Address of the physical screen)*

PHYSIC is a reserved variable which contains the location of the screen currently being displayed. If you load a different address into this variable, the screen will be immediately redrawn using the screen stored at the new address.

Example:

```
print physic
491520  (or 1015808 on a 1040ST)
10 reserve as screen 5
20 physic=5
30 cls
```

The above example reserves a memory bank as a screen and then assigns the address of this bank to the physical screen. Notice how you are able to use the number of the bank instead of an address.

When you run this program, the new screen will be cleared. If you now press the Undo key twice, the screen address will be returned to normal and the original picture will be restored. Incidentally, the ST's hardware will only let you display a screen stored at an address which is a multiple of 256 bytes. The RESERVE instruction automatically takes this into account when allocating memory for a screen.

LOGIC *(Address of logical screen)*

The Logical screen is the screen which is operated on by any of the text or graphics instructions. Normally this will be the same as the physical screen, but occasionally it's useful to use a separate screen to hold an image while it is being drawn. This allows you to draw one picture while displaying another, and then instantly switch between them using a special SCREEN SWAP instruction. A similar technique is used by games such as Starglider to generate impressive flicker free graphics. See SCREEN SWAP for a simple example of this process.

Example:

```
back=logic:rem Move the mouse around and see what happens.
print back
```

SCREEN SWAP *(Swaps the address of the logical and physical screens)*

Swaps the addresses of the physical and logical screens. This enables you to instantaneously switch the display between the two screens. Look at the example below.

```
10 cls
20 X1=50 : Y1=50 : X2=75 : Y2=100 : X3=25 : Y3=100
40 for I=0 to 244 step 8
50 ink 0
60 polygon X1+I-8,Y1 to X2+I-8,Y2 to X3+I-8,Y3 to X1+I-8,Y1
70 ink 1
80 polygon X1+I,Y1 to X2+I,Y2 to X3+I,Y3 to X1+I,Y1
100 next I
```

This program moves a triangle across the screen. As the triangle proceeds, it generates an intense and annoying flicker. You can solve this problem by displaying the triangle on the screen, only after it has been completely redrawn. Add the following lines to the program above:

```
30 logic=back
90 screen swap : wait vbl
```

You should also change:

```
60 polygon X1+I-16,Y1 to X2+I-16,Y2 to X3+I-16,Y3 to X1+I-16,Y1
```

Line 30 places the address of the sprite background into the logical screen. The triangle is now drawn on this screen without effecting the current image. The SCREEN SWAP instruction at line 90 then swaps the logical and physical screens around. This causes the finished version of the triangle to appear on the screen immediately.

The program now erases the old triangle from the invisible logical screen and redraws it at the next position. The whole process is subsequently repeated and the triangle apparently smoothly progresses from one side of the screen to the other. The reason for the change at line 60 incidentally, is simply to take into account the fact that each screen is used on alternate executions of the loop. This means that the triangle to be erased will be twice the distance from the current position as you would normally expect.

Note that we've intentionally exaggerated the flicker of the above example to illustrate the screen switching technique. In practice it would be very easy to reduce this problem considerably even without the use of the SCREEN SWAP

instruction. Also notice that as we've used the background screen for our own purposes, any of the sprite commands will interfere with the animation. Try moving the mouse while the program runs to observe this effect. Another example of screen switching can be found in the section on SCREEN COPY.

DEFAULT *(Return initial value of one of three screens)*

DEFAULT BACK	Returns initial value of *back*
DEFAULT PHYSIC	Returns initial value of *physic*
DEFAULT LOGIC	Returns initial value of *logic*

When you are using multiple screens, it's easy to lose track of the original screen addresses. The initial contents of the variables BACK, PHYSIC and LOGIC can be found at any time using the DEFAULT function. This function is often used at the end of a program to set the screen back to normal.

Examples:

```
physic=default physic
back=default back
logic=default logic
```

Do NOT confuse with the DEFAULT instruction.

Reserving a screen

As you have seen, any STOS Basic program can have a number of different screens in memory simultaneously. The following instructions allow you to allocate a memory bank to hold one of these screens.

RESERVE AS SCREEN *(Reserve a bank as a temporary screen)*

RESERVE AS SCREEN n

Reserves bank number *n* as a screen. The size of this bank is automatically set by RESERVE to 32768 bytes. After you have created a screen in this way, you can load it with data using either the LOAD instruction or SCREEN COPY.

Example:

```
10 reserve as screen 5
20 load "\stos\pic.pi1",5
```

Note that this screen is only intended for temporary storage and is reinitialised every time your program is run.

See RESERVE and LOAD.

RESERVE AS DATASCREEN *(Reserve a permanent screen)*

RESERVE AS A DATASCREEN n

The above command is identical to the RESERVE AS SCREEN instruction except for the fact that it is installed permanently into the ST's memory. Any screen you define as a DATASCREEN will be subsequently saved along with your program.

Example:

```
reserve as datascreen 5
clear
listbanks
```

See RESERVE (Chapter 3).

Loading a screen

STOS Basic lets you load a screen stored on the disc into either a memory bank or an address.

```
LOAD "IMAGE.NEO",scrn
LOAD "IMAGE.PI1",scrn
LOAD "IMAGE.PI2",scrn
LOAD "IMAGE.PI3",scrn
```

The LOAD command loads a screen into memory from the disc file IMAGE. An extension of NEO specifies that the file is stored in Neochrome format. Similarly, extensions of PI1,PI2,PI3 are used to signify a screen in Degas format. Note that *scrn* can be either a screen address, or the number of a memory bank.

Example:

```
10 load "\STOS\PIC.PI1",PHYSIC
20 wait key
30 default
```

Here is a larger example which converts screen files from Neochrome format to Degas format.

```
10 rem Neochrome to DEGAS converter
20 F$=file select$("*.NEO")
30 if F$="" then stop
40 reserve as screen 5
50 load F$,5
70 print "Press Return to save picture"
80 input "in DEGAS format";A$
90 right$(F$,3)="PI1"
100 save F$,5
110 input "Continue Y, or N";A$
120 if A$="y" or A$="Y" then 10
```

GET PALETTE *(Set the palette from a screen bank)*

GET PALETTE(n)

Loads the colour settings of a screen stored in bank *n*, and display them to the present screen.

Example:

```
10 reserve as screen 5
20 load "\STOS\PIC.PI1",5
30 physic=5
```

```
40 wait key
50 get palette(5)
60 wait key
```

CLS *(Clear the screen)*

In addition to the normal CLS instruction there is also an expanded version which enables you to erase sections of a screen stored anywhere in the ST's memory. There are three possible formats of this statement.

CLS scr	Clears the screen at *scr*
CLS scr,col	Fills the screen at *scr* with colour *col*
CLS scr,col,x1,y1 to x2,y2	Replaces the rectangle at *scr* at coordinates *x1,y1,x2,y2* with a block of colour col.

scr refers to either the address of a screen or the number of a memory bank. *col* can take any value from 0 to the maximum number of available colours. *x1,y,x2,y2* hold the coordinates of the top left and bottom right corners of the rectangle accordingly. This instruction provides a very fast and effective way of erasing parts of the screen.

Examples:

```
cls back:rem Erases the background screen

cls physic,6:rem Clears the physical screen with a block of colour 6

cls back,6,0,0 to 319,50:rem Erases the function key window from back
```

ZOOM *(Magnify a section of the screen)*

ZOOM scr1,x1,y1,x2,y2 TO [scr2,] x3,y3,x4,y4

Magnifies any rectangular section of the screen stored at *scr*1. *scr*1 and *scr2* can be either an address, or the number of a memory bank. The coordinates *x1,y1,x2,y2* refer to the size of the rectangular area which is to be enlarged.

x1,y1 denote the top left hand corner of this rectangle and *x2,y2* specifies the location of the corner diagonally opposite.

Similarly *x3,y3* and *x4,y4* hold the dimensions of the rectangle into which the screen segment will be expanded.

scr2 is an optional destination screen for the enlarged image. If it is not specified then the screen will be enlarged into the background held in *BACK*, and will then be copied into the current screen. This avoids any problems with the mouse or the sprites, and also displays the object in one smooth operation.

ZOOM is best suited to enlarging pictures with relatively large expanses of a single colour. This is because each individual point in the picture is magnified independently, which produces a noticable grain for large size increases.

An especially useful application of this instruction is in the creation of large banners on the screen.

Type in the example below:

```
10 rem ZOOM1
20 rem Set screen attributes
30 cls : mode 0 : pen 10 : curs off
40 Z$="Zooming!"
```

```
50 rem Find position of text
60 locate 0,4 : centre Z$
70 Y1=ygraphic(4) : X2=xgraphic(xcurs) : X1=X2-8*len(Z$) : Y2=Y1+8
80 for I=1 to 7
90 rem Calculate Zoom coordinates
100 X3=X1-16*I : Y3=Y1-16*I : X4=X2+16*I : Y4=Y2+16*I
110 rem Enlarge Text
120 zoom physic,X1,Y1,X2,Y2 to X3,Y2,X4,Y4
130 next I
140 wait key : curs on
```

This repeatedly enlarges the centred text starting at coordinates 0,4. We've kept the routine as general as possible to allow you to incorporate parts of it into your own programs.

We'll now expand this program slightly to demonstrate the page flipping mentioned earlier.

Add the following lines to the above program.

```
11 rem Reserve 6 screens
15 for I=5 to 11:reserve as screen I : cls I: next I
121 rem Enlarge text to screen no I
125 zoom physic,X1,Y1,X2,Y2 to I+5,X3,Y2,X4,Y4
140 rem Flip between all 6 screens
150 for I=6 to 11:physic=I:wait vbl : wait 5:next I
160 wait 30 : goto 140
```

You should also alter line 80 to

```
80 for I=1 to 6
```

Note that this program reserves six screens 32k long. It will work fine on a standard 520ST, providing you remove all STOS Basic accessories from memory using a line like:

```
accnew
```

In addition, you may also need to load STOS Basic directly on startup, rather than executing it from within Gem, as this saves you over 32k of memory.

Another common use of ZOOM is to magnify a specific part of an image for subsequent editing. The program below shows how this might be achieved in practice.

```
10 rem Zoom Example 2
20 mode 0
30 reserve as screen 5:rem Reserve a bank for the screen
50 F$=file select$("*.neo"):rem Choose a neochrome picture
60 if F$="" then stop
80 flash off:rem Turn off flashing
90 rem Load screen into Bank 5
100 load F$,5 : get palette (5)
110 rem Copy screen into Physical screen and Background
130 screen copy 5 to physic : screen copy 5 to back
140 rem Draw an expanding Box
150 gr writing 3
160 rem Click on the mouse to position Box
170 repeat : until mouse key : X1=x mouse : Y1=y mouse : X2=X1 : Y2=Y1
190 wait 40:rem Wait for Mouse key to be released
```

```
200 repeat
210 box X1,Y1 to X2,Y2
220 X2=x mouse : Y2=y mouse
230 box X1,Y1 to X2,Y2 : M=mouse key
250 until M<>0:rem click on a mouse button to exit
260 rem Make X1,Y1 into the top corner
270 if X1>X2 then swap X1,X2
280 if Y1>Y2 then swap Y1,Y2
290 rem If Right Mouse button pressed
300 rem Zoom Contents of Box to full
310 rem Screen.
320 if M=1 then zoom X1,Y1,X2,Y2 to 0,0,319,199 else box X1,Y1 to X2,Y2 :
M=0 : wait 40 : goto 170
330 wait key
340 goto 130
```

Much of this program should be self explanatory. Note the lines 140-250. These use the XOR writing mode to generate a simple expanding box. Feel free to use this routine in any of your own programs. After this box has been defined, the line at 320 uses the ZOOM command to expand its contents into the entire screen. Incidently, the test for *M=1* is merely to allow you to abort the current expansion by pressing the right mouse button.

REDUCE *(The inverse of zoom)*

REDUCE scr1 TO [scr2,]x1,y1,x2,y2

Compresses the entire screen stored at *scr1* into the box specified by the coordinates *x1,y1,x2,y2*. *x1* and *x2* hold the position of the top left corner of this box, and X2,Y2 the bottom right. *scr1* and *scr2* refer to either a screen address or the number of a memory bank. As with ZOOM, if the optional destination screen is omitted, the drawing is first placed in the background and then moved into the physical screen.

Example:

```
10 rem Reduce Example 1
20 F$=file select$("*.NEO")
30 rem Choose a picture
40 if F$="" then stop
50 mode 0 : flash off : curs off
60 rem Reserve screen and load Picture
70 erase 5:reserve as screen 5
80 load F$,5 : get palette (5)
90 rem display 4 copies of picture
100 for Y=0 to 1
110 for X=0 to 1
120 reduce 5 to X*160,Y*95,(X+1)*159+1,(Y+1)*96
130 next X
140 next Y
150 wait key
160 goto 20
```

This loads a Neochrome screen into a memory bank and then generates four smaller copies of it using the REDUCE at line 120.

If you've got the second example of ZOOM handy, you can change it to use the REDUCE instruction instead, with the line:

```
320 if M=1 then reduce 5 to X1,Y1,X2,Y2 else box X1,Y1 to X2,Y2 : M=0 : wait
40 : goto 170
```

REDUCE has many possible uses. One idea would be to generate a list of large icons similar to those utilised in the game STAR TREK. These could be assigned to a screen zone using SET ZONE, and then selected with the ZONE command. By storing a full-sized version in a compacted format (see PACK), you could then effectively expand these pictures into the entire screen.

SCREEN COPY *(Copy sections of the screen)*

SCREEN COPY scr1 TO scr2 (Copies *scr1* to *scr2*)

SCREEN COPY scr1,x1,y1,x2,y2 TO scr2,x3,y3

SCREEN COPY is undoubtably one of the most powerful of all the STOS Basic instructions. This is because it allows you to copy large sections of a screen from one place to another. As usual *scr1* and *scr2* can refer to either a screen address like *LOGIC* and *PHYSIC*, or the number of a memory bank. *x1,y1* and *x2,y2* hold the dimensions of the rectangular area which should be copied, and *x3,y3* contain the coordinates of the destination of this block. Note that the *x* coordinates used in this instruction are automatically rounded down to the nearest multiple of 16. Also the values taken by these numbers can be negative as well as positive. Look at the table below.

Graphics Mode	X Range	Y Range
Low	-320 to 320	-200 to 200
Medium	-640 to 640	-200 to 200
High	-640 to 640	-400 to 400

Any points in the destination outside the normal screen are simply not copied on the screen. This is in marked contrast with the BLIT statement supported by other versions of Basic which crash the ST completely if an illegal screen coordinate is used.

The best way to see how the various options work is by example. Before you can enter these examples you first need to do a little preparation. Start off by reserving a bank for the STOS Basic title screen with the line:

```
reserve as datascreen 10
```

Now place the STOS system disc into your drive and type:

```
load "\STOS\PIC.PI1",10 (for low resolution monitors)
```

or

```
load "\stos\pic.pi3",10 (for high resolution monitors)
```

Since you will be using the SCREEN COPY instruction rather a lot in this section, you can save yourself some typing by assigning it to one of the function keys like this:

```
KEY(10)="screen copy"
```

This allows you to abreviate any SCREEN COPY statements in subsequent listings to just f10.

Now copy the title in bank 10 into the logical screen using the lines:

```
cls
screen copy 10 to logic
```

As you move the mouse around on the screen, you will find that the picture will be steadily eaten away. This can be avoided by loading the picture into sprite background as well.

Example:

```
10 cls
20 screen copy 10 to logic
30 screen copy 10 to back
40 wait key
```

If you move the mouse when this program is being run, the screen will no longer be erased, because the sprite background now contains exactly the same picture as the logical screen.

By loading a picture into the background alone you can produce another interesting effect. Try typing:

```
cls
screen copy 10 to back
```

Now the title picture is steadily drawn as you move the mouse. Instant artwork! Now enter the lines:

```
delete 10-40: rem Do not type in NEW as this will erase bank 10
load "sprdemo.mbk"

10 cls : hide
20 screen copy 10 to logic
30 sprite 1,130,0,1
40 move y 1,"(1,1,1)L"
50 move on
60 wait key
```

Now for some more complicated examples. Type in the following lines:

```
screen copy 10,0,0,100,100 to logic,0,0
```

This copies the top left hand corner of the title on to the screen.

You can also use the SCREEN COPY statement with negative coordinates.

```
screen copy 10,0,0,100,100 to logic,-50,-50
```

As you can see, only the lower section of the block has been copied to the screen.

Here's one final example of the SCREEN COPY command which enables you to move a large coloured grid around on the screen using the mouse.

Example:

```
new
10 mode 0:l=14
```

```
15 rem Initialise screen and set square markers
20 cls physic : cls back:set mark 4,28
25 rem Draw a grid on the screen
30 for X=1 to 10 : for Y=1 to 9 : ink rnd(i)+1: polymark X*28,Y*20
40 next Y : next X
45 rem Reserve a screen and copy the grid to it
50 reserve as screen 10 : screen copy logic to 10
60 hide : curs off:rem Kill mouse and cursor
65 logic=back:rem Set Logical screen to sprite background
70 rem Move the grid
75 repeat
80 cls logic
85 rem Get mouse coords
90 X=320-x mouse*3 : Y=200-y mouse*3:rem Use different values for high
res
95 rem Copy the grid to the current screen
100 screen copy 10,X,Y,X+320,Y+200 to logic,0,0
110 screen swap:rem Swap physical and logical screens
120 wait vbl:rem Synchronise screen
130 until mouse key
140 default:rem Reset Editor window
```

The screen as a string

STOS Basic includes two special instructions which enable you to load a section of a screen into a string, and then manipulate it using the normal string commands. This data can then be copied anywhere on the screen using a single string assignment.

SCREEN$ *(Load an area of a screen into a string)*

There are two different forms of this statement.

s$=SCREEN$(scrn,x1,y1 TO x2,y2)

The SCREEN$ function is used to load an area of the screen bounded by the rectangle *x1,y1,x2,y2* into the string *s$*. *x1,y1* refer to the coordinates of the top left corner of this box, and *x2,y2* to the point diagonally opposite. Just as with the SCREEN COPY instruction, the X coordinates are automatically rounded down to the nearest multiple of 16. The expression SCRN can be either the address of a screen or the number of one of the memory banks.

Example:

```
A$=screen$(physic,0,0 to 319,199):rem Assigns the entire screen to a$

S$=screen$(back,50,50 to 100,100):rem A$=area from 50,50 to 100,100

reserve as screen 10
screen copy physic to 10
b$=screen$(10,0,0 to 160,100):rem Loads B$ with top of screen in bank 10
```

SCREEN$(scrn,x,y)=a$

This instruction copies a screen area from the string *a$* to the screen *scrn*, starting at the coordinates *x,y*. As usual *scrn* can refer to either a screen address or a bank

number. Also note that the *x* coordinates used by SCREEN$ are always rounded down to the nearest multiple of 16.

Warning! This command will only work with strings which have been previously loaded by the SCREEN$ function. The SCREEN$ statement provides you with a fast and efficient way of moving large objects around on the ST's screen.

Examples:

```
10 S$=screen$(physic,0,0 to 100,100)
20 for y=0 to 3:for x=0 to 6
30 screen$(physic,50*x,50*y)=S$
40 next x:next y
```

This example fills the screen with copies of the top corner of the display.

The classic application of SCREEN$ is in the creation of complex backgrounds for your games. By building your picture out of a number of previously defined blocks, you can combine these into a wide range of different screens. Furthermore, after you have stored your blocks into memory, you can hold each screen as a simple list of numbers. In practice this simple technique can save you an immense amount of space.

Example:

```
5 rem SCREEN$ example
6 rem Requires Disc containing complete \STOS\ folder in order to run.
10 dim P$(10,6)
15 rem Use extension PI3 for MONO MODE.
20 mode 0 : curs off : hide :load "\STOS\PIC.PI1",back
30 for X=0 to 9
40 for Y=0 to 5
45 rem Copy screen segments into array
50 P$(X,Y)=screen$(back,X*32,Y*32 to (X+1)*32,(X+1)*32)
60 next Y
70 next X
80 for X=0 to 9
90 for Y=0 to 5
100 X1=rnd(9):Y1=rnd(5)
105 rem Copy segments back onto screen
110 screen$(physic,X*32,Y*32)=P$(X1,Y1)
120 next Y
130 next X
140 wait key
150 goto 80
```

In order to make it as easy as possible to draw one of these screens we have provided you with a special MAP DEFINER program.

Scrolling the screen

DEF SCROLL *(Define a scrolling zone)*

DEF SCROLL n,x1,y1 to x2,y2,dx,dy

DEF SCROLL allows you to define up to 16 different scrolling zones. Each of these is associated with a specific scrolling operation determined by the variables *dx* and

dy. *n* denotes the number of the zone and can range from 1-16. *x1,y1* refer to the coordinates of the top left hand corner of the area to be scrolled, and *x2,y2* to the point diagonally opposite.

dx signifies the number of pixels the zone will be shifted to the right in each operation. Negative numbers indicate that the scrolling will be from right to left, and positive numbers from left to right.

Similarly, *dy* holds the number of points the zone will be advanced up or down during the scroll. In this case negative values of *dy* are used to indicate an upward movement and positive values a downward one.

SCROLL *(Scroll the screen)*

SCROLL n

The SCROLL command scrolls the screen in the direction you have previously specified with the DEF SCROLL instruction. *n* refers to the number of the zone you wish to scroll.

Example:

```
10 def scroll 1,0,0 to 320,200,1,0
20 scroll 1:goto 20
```

Do NOT confuse with the SCROLL instruction used by the window commands.

Now for a larger example:

```
5 rem Vertical Scrolls
10 input "Step Size?";S:rem Choose scroll increment
11 rem Initialise screen and load background from system disc
20 mode 0 : curs off : hide : load "\STOS\PIC.PI1",back
30 def scroll 1,80,0 to 240,200,0,-S:rem Define scrolling zone 1
40 for Y=0 to 199 step S:rem Scroll section of the screen
45 rem copy top of screen to bottom
50 screen copy back,80,Y,240,Y+S to logic,80,200-S
60 scroll 1:rem scroll zone 1
70 next Y
80 goto 40
```

This loads an image from the STOS system disc and rotates it around on the screen. The variable *S* holds the number of points the picture will be moved when each SCROLL instruction is executed. The larger the value of S, the faster and jerkier the scrolling. Note line 50. This copies the top section of the screen into the bottom before it disappears.

Here is another example which demonstrates how horizontal scrolling can be achieved.

```
5 input "Speed";S
7 rem Initialise screen and load background from system disc
10 mode 0 : curs off : hide : load "\STOS\PIC.PI1",back
20 def scroll 1,0,80 to 320,120,-16,0:rem Define
scrolling zone 1
30 for Y=0 to 319 step 16:rem Scroll section of the screen
35 rem Copy left section of the screen back to the right
40 screen copy back,Y,80,Y+16,120 to logic,320-16,80 : for W=1 to S : next W
: scroll 1
50 next Y
60 goto 30
```

This uses a very similar technique to the last example except for the fact that SCREEN COPY rounds all X coordinates down to the nearest multiple of 16. The example is therefore forced to scroll in units of 16. Despite this the scrolling is still reasonably smooth, especially at the slower speeds.

Now for a final example which combines a complex series of scrolling zones to produce a fascinating effect on the screen.

```
1 rem Screen Scrolling demo
5 rem Needs Stos system disc in drive
10 mode 0 : curs off : hide : load "\stos\pic.pi1",back
15 rem Define scrolls
20 def scroll 1,0,171 to 320,200,0,-6
30 def scroll 2,0,146 to 320,175,0,-4
40 def scroll 3,0,122 to 320,150,0,-2
50 def scroll 4,0,72 to 320,125,0,-1
60 def scroll 5,0,46 to 320,75,0,-2
70 def scroll 6,0,21 to 320,50,0,-4
80 def scroll 7,0,0 to 320,25,0,-4
90 rem scroll screen
100 for Y=0 to 199
110 screen copy back,0,Y,320,Y+6 to logic,0,194
130 scroll 1 : scroll 2 : scroll 3 : scroll 4 : scroll 5 : scroll 6 : scroll 7
140 next Y
150 goto 100
```

Screen synchronisation

Like most microcomputer systems, the Atari ST uses a memory-mapped display. This is a technical term for a concept you are almost certainly already familiar with. Put simply, a memory-mapped display is one which uses special hardware to convert an image stored in memory into a signal which can be displayed on your television screen. Whenever STOS Basic accesses the screen it does so through the medium of this screen memory.

The screen display is updated by the hardware every 50th of a second (70th in Monochrome mode). Once a screen has been drawn the electron beam turns off and returns to the top left of the screen, this process is called the vertical blank or VBL for short. At the same time, STOS Basic performs a number of important tasks, such as moving the sprites and switching the physical screen address if it has changed. The actions of instructions such as PUT SPRITE, or SCREEN SWAP will therefore only be fully completed when the screen is next drawn. Since a 50th of a second is quite a long time for STOS Basic, this can lead to a serious lack of coordination between your program and the screen, which is especially noticable when the next instruction also manipulates the screen in some way. The only effective method of avoiding this difficulty is to wait until the screen has been updated before you execute the next Basic command.

WAIT VBL *(Wait for a vertical blank)*

The WAIT VBL instruction halts the ST until the next vertical blank is performed. It is commonly used after either a PUT SPRITE instruction, or a SCREEN SWAP. As a general rule, if your program uses sprites or screens and it only works intermittantly, it's always worth checking to see whether you have omitted the WAIT VBL.

SYNCHRO *(Synchronise scrolling with sprites)*

STOS Basic performs all sprite movements every 50th of a second. This generally

works fine, but occassionally it leads to an irritating synchronisation problem.

Supposing you want to place a sprite at a fixed point on a scrolling background. Whenever this background moves, the sprite will move along with it. It would be easy enough to produce a set of MOVE X and MOVE Y instructions which precisely followed the movement of the background. Unfortunately, this wouldn't quite work as the SCROLL instructions would not be executing at the same time as the sprite movements. The sprite would therefore tend to drift jerkily around on the screen.

Luckily, STOS Basic includes a useful SYNCHRO instruction which allows you to move all the sprites on the screen at the exact moment you require. This enables you to effortlessly synchronise the sprites with a scrolling background.

There are three forms of this instruction:

SYNCHRO OFF	Turns off the normal sprite interrupt which moves the sprites every 50th of a second.
SYNCHRO	Executes all the sprite movements exactly once.
SYNCHRO ON	Reverts the sprite movements to normal. The sprites will now be moved in the normal way every 50th of a second.

We'll demonstrate how all this actually works with a small example.

First you need to load some sprites into your micro. Place the accessory disc into the drive and type:

```
load "sprdemo.mbk"
```

You can now type in the program itself:

```
new
10 rem Demonstration of SYNCHRO
20 mode 0 : curs off : hide : key off
30 rem Load picture from disc
40 load "\STOS\PIC.PI1",back : screen copy back to logic
50 rem Place sprite on the screen
60 rem Start it moving up.
70 sprite 1,144,199,1 : move y 1,"(1,-2,1)L"
80 rem Turn off sprite interrupt
90 synchro off : move on
100 rem Define Scrolls
110 def scroll 1,80,0 to 240,200,0,-2
120 rem Scroll section of the screen
130 wait 100 : rem Wait for drive to stop
140 for Y=0 to 199 step 2
150 screen copy back,80,Y,240,Y+2 to logic,80,198
160 scroll 1 : wait vbl : synchro
170 next Y
180 rem Restart from bottom of screen
190 sprite 1,144,199,1 : move y 1,"(1,-2,1)L"
200 synchro off : move on
210 goto 140
```

Notice line 160 which moves the sprite up one unit and then scrolls the screen along with it. The WAIT VBL instruction is essential as it completes the syncronization process. Try removing it and see what happens.

I've chosen this specific sprite to illustrate an interesting side effect. As the

sprite is moved, this specific sprite background peeps through it, rather like a window. You could use this technique to produce a range of useful special effects.

Compacting the screen

STOS Basic comes complete with a useful accessory which allows you to compact any screen files stored in either Neochrome or Degas format into just a fraction of their normal size. You can load this program from the accessory disc using the line:

```
accnew:accload "compact.acb"
```

Using the compactor is simplicity itself. You start off by clicking on one of the LOAD FILE options. This presents you with a standard STOS file selector which can be used to choose a file in the normal way. The screen you have selected is now loaded into the ST's memory and displayed. To return to the main menu just press the left mouse button once.

If you wish to compact the whole screen, choose the PACK WHOLE SCREEN option from the Picture menu. The compactor will now attempt to compress the screen using a number of different strategies. As soon as it finds the one which uses the smallest amount of space, it will compact the file. This file can be saved either as a memory bank or a raw binary file. The easiest option to use is the memory bank, as this lets you subsequently load the screen directly into STOS Basic. You can also use the Quit and Grab option to incorporate the screen straight into your current Basic program.

In order to compact only part of the screen you begin by selecting the appropriate option from the Picture menu. Although this section does include a comprehensive set of instructions, we'll summarise them here for completeness.

1. Click on a mouse button to display the whole picture.

2. You start by choosing the left hand corner of the area to be compacted by clicking on the left button. If you now press the right button and move the mouse, an expanding box will be drawn. This box encloses the section of the screen you have currently chosen. Similarly, by pressing the left hand button again, you can change the position of the top corner of this rectangle.

3. After you have selected part of the screen to be compressed, press the spacebar to compact your image. You can now save this picture on the disc using the Disc menu as before.

The compaction utility would be useless if there was not some easy way of restoring the screen to its full size. This can be done using the UNPACK instruction.

UNPACK *(Unpack a screen compacted with the accessory)*

UNPACK bnk,scr

The UNPACK command restores a compacted screen stored in bank number *bnk* into the screen *scr*. As usual *scr* can refer to either a bank defined as a SCREEN or DATASCREEN, or a screen address.

Example:

```
load "backgrnd.mbk:rem Load a compressed screen saved in bank 5
```

```
unpack 5,back:rem Unpack bank five and load into sprite background
physic=back:rem Set physical screen to sprite background
```

PACK *(Function to pack a screen)*

l=PACK scr,bnk

This is just the reverse of the UNPACK command. It's normally easier to use the SCREEN COMPACTOR accessory, but if you do need to compact a screen within a program, you can use the PACK function. *scr* refers to either a screen address or a bank number containing a screen to be compressed. *bnk* denotes the bank which is to be used as a destination. After the pack function has been executed, *l* is loaded with the length of the compressed screen.

Example:

```
reserve as screen 5:rem Reserve space for source
reserve as screen 6:rem Reserve space for destination
load "\stos\pic.pi1",5:rem Load Title screen from
system disc in 5
L=pack(5,6):rem Pack screen
reserve as data 7,L:rem Reserve space for new screen
copy start(6),start(6)+l to start(7)
save "title.mbk":rem Save compacted screen
```

Special screen effects

APPEAR *(Fade between two pictures)*

APPEAR x [,y]

The APPEAR command enables you to produce fancy fades between a picture stored in address *x* or in bank *x*, and the current screen. The *y* value is optional and refers to the type of fade you wish to use. *y* can range from 1 to 79. Fades between 1-72 always result in a COMPLETE image being copied from *x* to the screen. Fades from 73-79 leave the final screen slightly different from the original in bank *x*.

Type in the example below placing your backup of the STOS system disc into the current drive.

Example:

```
10 hide
20 reserve as screen 15
30 if mode=1 then mode=0
40 if mode=0 then load "\stos\pic.pi1",15 else load "\stos\pic.pi3",15
50 cls
60 input "screen effect";X
70 curs off
80 if X=0 then default : end
90 get palette (15)
100 appear 15,X
110 wait key
120 curs on
130 goto 50
```

FADE *(Blend one or more colours to new colour values)*

This function allows you to produce stunning effects in one simple command. There are three formats of the FADE command:

FADE speed	Fade all colours to black This version of FADE reduces each colours RBG vaues by 1 until they reach zero. *speed* is the amount of vertical blanks that must occur before another change to the palete is made.
FADE speed TO sbank	Fade the present colours to those of the specified screen The current colours are blended into the palette of the screen stored in bank *sbank*.
FADE speed,col1,col2,	FADE separate colours to a new value This is the most powerful of the three formats and allows any colour to be blended into another. Enter the line:

```
10 mode 0:print "bye bye...":fade 3:wait 7*3
```

The WAIT command is used after the FADE because the fading changes are done during interrupt. Thus the program carries on. Because our next line will reset the colours, it's best to wait until the original fade has been completed. The pause value for the WAIT command can be calculated by the formula:

```
wait value = fade speed * 7
```

Once the above line has been run, the screen is left in total darkness. To bring back some colour you would enter a line like:

```
20 cls:print "here I am again!":fade 3,,$777,$700
```

Notice that there are two commas after the speed parameter. This tells STOS Basic that you don't wish to change the value of colour 0 and this can be applied to any colour in the palette. Colours 1 and 2 are now faded up to reveal the new message.

Fade adds flare to your programs and gives them a professional touch similar to credit screens from films.

Examples:

```
fade 3:rem press undo twice to see again

reserve as datascreen 15
load "\STOS\PIC.PI1",15
fade 10 to 15

fade 5,$777,$777,$777,$777,$777,$777,$777,$777,$777,$777,
$777,$777,$777,$777,$777,$777
```

Pattern Setting

SET PATTERN *(Set up the fill pattern)*

SET PATTERN a$

You can set up a user defined fill pattern with this command. *a$* must contain the fill definition which must be a 16x16 block.

Example:

```
A$=screen$(physic,1,1 to 16,16)
set pattern A$
```

This is in addition to the other SET PATTERN format.

See PAINT, SCREEN$

8 Text and windows

STOS Basic allows you to print text on the screen in a number of different ways. Up to 13 windows can be displayed at any one time, and each of these can have its own unique set of characters.

Text attributes

Every STOS Basic window has a separate set of attributes, such as the character and background colours of the enclosed text.

PEN *(Set colour of text)*

PEN index

The PEN instruction allows you to specify the colour of any text which will subsequently be displayed in the current window. This colour can be chosen from one of up to 16 different colours. As you might expect, the number of colours available varies between the different graphics modes.

Mode	Allowable index numbers
0 (Low)	0-15
1 (Medium)	0-3
2 (High)	0-1

Example:

```
new
10 mode 0
20 for I=0 to 15
30 pen I
40 print "Pen number ";I;space$(10)
50 next I
60 pen 1
```

As a default, the pen colour is set to index number 1.

See COLOUR, PALETTE, PAPER.

PAPER *(Set colour of background of text)*

PAPER index

PAPER designates a colour to be used as the background for the text. As with PEN, *index* denotes a colour number from 0-15 (0-3 in medium res).

Example:

```
new
10 mode 0
```

```
20 for I=0 to 15
30 paper I
40 print "Paper number ";I;space$(10)
50 next I
60 wait key
70 default
```

On startup the background of a window is set to colour 0.

See PEN, COLOUR, PALETTE.

INVERSE ON/OFF *(Enter inverse mode)*

INVERSE ON swaps the text and background colours specified by PEN and PAPER. The effect of this is to invert any new text which is printed on the current window.

Example:

```
new
10 print "This is some text in normal mode"
20 inverse on
30 print "This is some inverted text"
40 inverse off
```

See SHADE, UNDER, WRITING.

SHADE ON/OFF (*Shade all subsequent text*)

SHADE highlights any new text on a window by reducing the brightness of the characters with a mask.

Example:

```
new
10 mode 1
20 print "Normal Text"
30 shade on
40 print "Shaded Text"
50 shade off
```

See UNDER, INVERSE, WRITING.

UNDER ON/OFF *(Set underline mode)*

This instruction causes the text in the current window to be underlined.

Example:

```
UNDER ON
? "UNDERLINED"
UNDERLINED
UNDER OFF
? "NORMAL"
NORMAL
```

See SHADE, INVERT, WRITING.

WRITING *(Change text writing mode)*

WRITING effect

The WRITING command allows you to change the writing mode used for all future text output.

Writing mode effect:
1 Replacement mode (Default)
2 OR mode. All characters are merged on the screen with a logical OR.
3 XOR mode. Characters combined with background using XOR.

Example:

```
new
5 mode 0
10 bar 0,0 to 319,199
20 print "Normal text"
30 writing 2
40 print "OR mode"
50 writing 3
60 print "XOR mode"
70 wait key
80 default
```

Do NOT confuse with GR WRITING.

Cursor functions

Any text you output to the screen using the PRINT instruction is always printed at the current cursor position. STOS Basic includes a range of facilities which allow you to move this cursor around, and print text practically anywhere on the screen.

LOCATE *(Position the cursor)*

LOCATE x,y

LOCATE sets the current cursor position to the coordinates *x* and *y*. This sets the starting point for all future text operations on the screen. LOCATE uses a special type of coordinates known as text coordinates. These are measured in units of a single character, relative to the top left hand corner of the current window. So the coordinates 10,10 refer to a point 10 characters down from the top of the window, and 10 characters across from the left.

Example:

```
locate 10,10:print "Hi"
```

The possible range of these coordinates varies depending on the dimensions of the window you are using, and the size of the character set.

Here is a small table showing the size of the screen in text coordinates in each of the three graphics modes.

Mode	X range	Y range
0	0-39	0-24
1	0-79	0-24
2	0-79	0-24

Conversion functions

STOS Basic provides you with a useful set of four functions which readily enable you to convert between these text and graphic coordinates.

=XTEXT *(Convert an x coordinate from graphic format to text)*

t=XTEXT(x)

This function takes a normal X coordinate ranging from 0-639 (0-319) in low res) and converts it to a text coordinate relative to the current window. If the screen coordinate lies outside the window then a negative value is returned. The following example should make this a little clearer:

```
new
10 cls:print "Move the mouse about!"
20 repeat
30 X=xtext(x mouse) : if X<0 then 60
40 Y=ytext(y mouse) : if Y<0 then 60
50 locate X,Y : print "*":rem Print * at current mouse pointer.
60 until mouse key:rem Exit when a mouse button is clicked.
70 default
```

See YTEXT, LOCATE, WINDOPEN, XGRAPHIC, YGRAPHIC

=YTEXT *(Convert a y coordinate from a graphic format to text)*

t=YTEXT(y)

YTEXT converts a coordinate ranging from 0-199 (0-399 in high res) into a text coordinate relative to the current window.

See XTEXT for more details. Also YGRAPHIC, XGRAPHIC, LOCATE.

=XGRAPHIC *(Convert an x coordinate from text format to graphic)*

g=XGRAPHIC(x)

The XGRAPHIC function is effectively the inverse of XTEXT, in that it takes a text coordinate ranging from 0 to the width of the current window and converts it into an absolute screen coordinate.

Example:

```
new
5 mode 0 :ink 1
10 windopen 1,3,3,30,10
20 print xgraphic(0),ygraphic(0)
30 draw xgraphic(0),ygraphic(0) to xgraphic(27),ygraphic(7)
40 wait key
50 windel 1
```

Note that there's also an equivalent function for Y coordinates called YGRAPHIC.

See XTEXT, YTEXT, YGRAPHIC.

=YGRAPHIC *(Convert a y coordinate from text format to graphic coordinate)*

g=YGRAPHIC(y)

This function converts a coordinate in text format relative to the current window into an absolute screen coordinate.

See XGRAPHIC, XTEXT, YTEXT.

SQUARE *(Draw a rectangle at the current cursor position)*

SQUARE wx,hy,border

SQUARE draws a rectangle *wx* characters wide by *hy* characters high at the cursor position. *border* can be any of the 16 possible border types used by the windows. See BORDER for more details. *wx* and *hy* can range from 3 to the size of the current window. After this instruction has been executed, the text cursor is placed at the top left corner of the new box.

Example:

```
10 square 10,10,3
20 print "Square "
```

Now for a slightly larger example, which shows off all the 15 different border types:

```
10 cls
20 for I=1 to 15
30 locate I*2,20-I
40 square I+3,I+3,I
50 next I
60 goto 60
```

See BORDER, XTEXT, YTEXT

HOME *(Cursor home)*

HOME moves the text cursor to the top left hand corner of the current window (coordinates 0,0).

Example:

```
10 cls
20 locate 10,10
30 print "Demonstration of "
40 home
50 print "HOME"
```

See LOCATE, XCURS, YCURS.

CDOWN *(Cursor down)*

CDOWN pushes the text cursor down one line. The same effect can also be achieved using the line:

```
print chr$(10)
```

Example:

```
print "Example":cdown:cdown:print "of cdown"
```

See CUP, CLEFT, CRIGHT.

CUP *(Cursor up)*

CUP moves the text cursor up by a line, in the same way that CDOWN shifts it down. This instruction is logically identical to the line:

```
print chr$(11);
```

Example:

```
print "Example":cup:cup:print "of cup"
```

See CLEFT, CDOWN, CRIGHT.

CLEFT *(Cursor left)*

The CLEFT instruction displaces the text cursor one character to the left. Note that CLEFT is equivalent to PRINT CHR$(3).

Example:

```
print "Example":cleft:cleft:print "of cleft"
```

See CUP, CRIGHT, CDOWN.

CRIGHT *(Cursor right)*

CRIGHT has the opposite effect as CLEFT and moves the cursor one place to the right. An identical effect can be achieved using the line:

Example:

```
print chr$(9)
print "Example":cright:cright:print "of cright"
```

XCURS *(Variable holding the X coordinate of the text cursor)*

XCURS is a variable which returns the X coordinate of the text cursor (in text format).

Example:

```
locate 10,0:print XCURS
10
```

YCURS *(Variable holding the Y coordinate of the cursor)*

YCURS returns the Y coordinate of the text cursor (in text format).

Example:

```
locate 0,10:print ycurs
10
```

SET CURS *(Set text cursor size)*

SET CURS top,base

The SET CURS instruction allows you to change the size of the text cursor. *top* refers to the topmost point of the cursor, and *base* to the bottom. These values can range from 1 to the maximum height of a character (normally 8 in medium and low resolution).

Example:

```
set curs 1,8
```

CURS ON/OFF *(Enable/disable text cursor)*

This function removes the flashing cursor from the current window. In order to stop the cursor flashing, CURS OFF deactivates colour number 2. Since the action of colour 2 is not restricted to a single window, any pictures drawn in this colour will immediately cease flashing. Similarly, the flashing cursors in every other window will also be frozen.

Text input/output

CENTRE *(Print a line of text centred on the screen)*

CENTRE a$

CENTRE takes the string in *a$* and prints it in the centre of the screen. This text is printed on the line currently occupied by the text cursor.

Example:

```
new
10 locate 0,1
20 centre "This is a centered TITLE"
30 locate 0,3
40 centre "And this is another one"
```

TAB *(Move the cursor to the right)*

TAB(n)

TAB is often used in conjunction with the PRINT instruction to space out a line of text on the screen. The action of the TAB is to move the text cursor *n* places to the right before the next print operation. It does this by generating a string of CHR$(9) characters.

Example:

```
print tab(10);"Example: of TAB"
```

Example of TAB

Also:

```
X$=tab(15)
print X$;"15 spaces to the right"
```

15 spaces to the right

See PRINT, CRIGHT.

SCRN *(Return the character on the screen at a specific coordinate)*

SCRN(x,y)

SCRN is a function which returns an Ascii character to be found at the text coordinates *x* and *y* relative to the current window.

Example:

```
new
10 locate 0,0
20 print "Hello"
30 locate 0,10
40 for I=0 to 5
50 print chr$(scrn(I,0));" ";scrn(I,0)
60 next I
```

See LOCATE, PRINT.

Windows

WINDOPEN *(Create a window)*

The WINDOPEN instruction enables you to create a window on the ST's screen. There are three possible formats to this statement.

WINDOPEN n,x1,y1,w,h
WINDOPEN n,x1,y1,w,h,border
WINDOPEN n,x1,y1,w,h,border,set

n is the number of the window to be opened. Permissible values for *n* range from 1-13.

x1,y1 are the text coordinates to the top left hand corner of the new window.

w,h specify the size in characters of the new window. Note that the minimum size of these windows is 3 by 3.

Border chooses one of 16 possible border styles for the new window. See BORDER for more details.

Set indicates which character set is to be used. This takes the form of a number which can range from 1 to 16 depending on the sets currently installed in the ST's memory. The default values for the sets from 1 to 3 are:

Set	Size	Notes.
1	8x8	pixels default set for low resolution
2	8x8	pixels default set for medium resolution
3	8x16	pixels default set for high resolution

You can happily use all of these sets in each of the three resolutions. Set three in

particular can be especially effective on a colour monitor as it provides you with a useful set of large characters.

Note: the text coordinates x1,y1 and the window size w,h use the new character sizes! You can also use the font definition accessory to create your own character sets. These sets are given numbers ranging from 4-16. See the separate section on character sets for more details.

Example:

```
new
10 windopen 1,1,1,39,20 : rem Open a large window
20 windopen 2,10,10,20,5,10 : rem Small window with border 10
30 windopen 3,20,15,20,4,0,1 : rem Open a window using character set one
40 windopen 4,3,10,30,5,3,2 : rem Window with set 2 and border 3
50 windopen 5,10,3,20,5,5,3 : rem Window with set 3 and border 5
```

In order to test these windows you can use the WINDOW function like so:

```
window 2
window 4
window 1
window 3
window 5
```

Here's another example which opens five windows on the screen, each with its own separate set of attributes.

```
5 mode 0
10 for I=1 to 5
20 windopen I,1,1+(I-1)*5,39,4,I
30 paper I : ink I+10
40 print "Window ";I;" "
50 next I
```

As before, you can flick between these windows using *window*:

```
window 3
```

See WINDEL, WINDOW, QWINDOW, WINDCOPY, WINDON, WINDMOVE, Character sets.

TITLE *(Define a title for the current window)*

TITLE a$

The TITLE instruction sets the top line of the current window to the title string in *a$*. If the length of this string is less than the width of the window, then it is centred. This title will now be displayed along with the window, until it is deleted by using the BORDER command with no parameter.

Example:

```
new
5 mode 0
10 windopen 5,1,1,20,10
```

```
20 title "Window number 5"
30 wait key
40 border
50 wait key
60 windel 5
```

See BORDER, WINDEL, WINDOPEN, WINDMOVE, WINDOW.

BORDER *(Set the border of the current window)*

BORDER n

This instruction allows you to choose from one of 16 possible borders for the current window. The variable *n* can take values ranging from 1 to 16. These borders are made up from the Ascii characters 192 to 255 and can be readily changed using the FONTS.ACB accessory.

Example:

```
new
default
10 windopen 5,5,5,20,10
20 title "Window number 5"
30 wait key
40 for I=1 to 16:border I:wait 5:next I
50 windel 5
```

Note that if you use the BORDER command on its own, the current border is redrawn, and any title associated with the current window is erased.

WINDOW *(Activate window)*

WINDOW n

WINDOW sets the current window to window number *n*. It then redraws the window along with any of its contents. This instruction should normally only be used when a number of windows overlap on the screen. If this is not the case then it makes rather more sense to use the QWINDOW statement which activates the window without redrawing it as this command is much faster than WINDOW.

Example:

```
new
10 for I=1 to 13
20 windopen I,I+5,I+2,20,8
30 next I
```

Now type in the lines:

```
run
window 5
window 10
```

Press undo twice to revert the screen to normal.

See QWINDOW, WINDEL, WINDOPEN, WINDON, WINCOPY

QWINDOW *(Activate window without redrawing it)*

QWINDOW n

This function sets the current window to window number *n*, but does not redraw the window. It should therefore only be used if you're absolutely sure that the window has not been overwritten by something else.

Example:

```
new
10 for I=1 to 5
20 windopen I,1,I*4,15,4 : windopen I+5,20,I*4,15,4
30 next I
run
qwindow 1
qwindow 5
qwindow 8
```

Note that because QWINDOW does not have to redraw the contents of the window, it is considerably faster than the equivalent WINDOW command. Further examples of this instruction can be found in the accessories supplied with the package. These can be examined using SEARCH:

```
load "FONTS.ACB"
search "qwindow"
```

WINDON *(Variable containing number of the current window)*

WINDON returns the number of the currently active window.

Example:

```
new
10 windopen rnd(12)+1,10,10,10,10
20 print "Window number ";windon," Activated"
```

See WINDOW, QWINDOW, WINDOPEN.

WINDMOVE *(Move a window)*

WINDMOVE x1,y1

WINDMOVE moves both the current window and its contents to a new part of the screen specified by the text coordinates *x1,y1*. These coordinates are based on the character size of the window which is to be moved.

Example:

```
WINDOPEN 1,0,2,30,10
WINDMOVE 5,3
```

See WINDOW, QWINDOW, WINDON, WINDOPEN.

WINDEL *(Delete a window)*

WINDEL n

This function deletes the window number *n*, and erases it from the screen. If the window to be deleted is the current window, then the current window will be set to the window with the next lowest number, and this will be redrawn automatically.

Example:

```
new
10 for I=1 to 13
20 windopen I,I+5,I+2,10,10
30 next I
40 for I=1 to 13
50 wait key
60 windel I
70 next I
```

See WINDOPEN, WINDMOVE, WINDOW, QWINDOW, WINDON, WINDCOPY.

CLW *(Clear the current window)*

CLW erases the contents of the current window and replaces it with a block of the current PAPER colour. Note that you can perform a CLW instruction from the editor by pressing the Clr key (or Shift+Home).

Example:

```
clw:rem Clears window 0.
```

SCROLL ON/OFF *(Switch window scrolling on and off)*

The SCROLL instruction is used to control the scrolling of the current window.

SCROLL OFF turns off the scrolling. Whenever the cursor reaches past the bottom of the screen it will now reappear from the top.

SCROLL ON restarts the scrolling. A new line is now automatically inserted when the cursor attempts to reach past the bottom of the screen.

Example:

```
scroll off
```

Do NOT confuse this function with DEF SCROLL!

See SCROLL UP, SCROLL DOWN.

SCROLL UP *(Scroll the current window up)*

This instruction moves a section of the current window above the text cursor, one line up. Anything on the top line of the window is erased.

Example:

```
scroll up:scroll up:scroll up
```

Not to be confused with DEF SCROLL.

See SCROLL DOWN, SCROLL.

SCROLL DOWN *(Scroll the current window down one line)*

SCROLL DOWN scrolls the area below the text cursor one line down. As a natural consequence of this instruction, the bottom line of the window will be overwritten.

Example:

```
scroll down:scroll down:scroll down
```

See SCROLL UP, SCROLL.

Character sets

Each STOS Basic window can have its own individual character set. Three of these sets are provided on the disc as standard, and these can be edited or changed using the character definer FONTS.ACB.

In order to build your own character set, you should first load the font accessory FONTS.ACB. Load this by inserting the STOS accessory disk and typing in the line

```
accnew: accload "FONTS.ACB"
```

You can access this at any time by pressing the keys Help+f1. When this utility is executed, the screen consists of a drop-down menu, along with two windows. The leftmost of these windows is used to edit a character, and the rightmost window is used to select the character to be redefined.

Start off by moving the mouse pointer to the selection window. Notice how the character underneath the mouse pointer is inverted, and its Ascii code is displayed at the bottom of the screen. This character can be chosen by clicking the left mouse button.

You can now edit your character by moving the mouse cursor into the edit window, and clicking on either the left or the right mouse buttons. The left button sets a point at the current cursor position, and the right button erases it.

In addition, you can also manipulate your character using one of the many options from the tool and draw menus.

After you have finished drawing your new character you can install it into the current set by moving the mouse back to the selection window, and positioning the pointer onto the character you wish to change. This character can now be overwritten with the new data by clicking on the right mouse button.

The final step in the creation of the character set is to save it. There are two possible alternatives. Firstly you can save the set to the disc in a file with the extension .MBK. This file can then be loaded at a later date. You can also load your set directly into your current program using the Quit & Grab option. This places the new character set into bank five, and then exits back to the STOS Basic editor.

Here is a summary of the entire process:

1. Choose a character from the Selection window using the left button.
2. Edit the character in the Edit window. The left button sets a point. The right button deletes a point. The Tool and Draw menus manipulate the character.
3. Install the character in the Selection window with the right mouse button.
4. Repeat stages one to three until you have completed your new character set.
5. Save the set using either the Save or the Quit & Grab options from the Disc menu.

The System menu allows you to select one of four possible sizes for your characters. Unfortunately, not all of these options are available in all three graphics

modes. Look at the following table.

Size	Modes allowed
8x8	All.
8x16	High and medium resolutions
16x8	High resolution only
16x16	High resolution only

Before you can call a user-defined character set, you first need to reserve some space and load this set into memory. This is done automatically by the Quit & Grab option from the font definer. If you intend to install a number of sets, it's easiest to save the sets to the disc, and then incorporate them into your program by hand.

Saving space

RESERVE AS SET *(Reserve a bank of memory for a character set)*

RESERVE AS SET n,len

This reserves *len* bytes of space in bank number *n* for a character set. This set can now be loaded into the bank using a line like:

```
LOAD "FONT1.MBK",n
```

Example:

```
reserve as set 5,4000
load "FONT1.MBK",5
```

Note that the bank defined using this command is permanent and will be automatically included with your current program when you save it to the disc. The file FONT1.MBK is one of three example character sets supplied with the package. Each additional set is given a unique number ranging between four and nine. The first character set you defined is denoted by the number four, the second by five and so on.

Supposing, for example, you reserve some space for three character sets like so:

```
RESERVE AS SET 6,4000
RESERVE AS SET 8,4000
RESERVE AS SET 5,4000
```

These sets would be accessed using the numbers: 4 for bank 6,5 for bank 8,6 for bank 5. The size of these banks has been set to 4,000 bytes.

You can calculate how large a character set is using the CHARLEN function.

CHARLEN *(Get the length of a character set)*

CHARLEN (n)

This function returns the length of a character set specified by the number *n*. Numbers one to three represent the system sets, and numbers 4 to 16 represent supplementary sets created using FONTS.ACB.

Example:

```
? charlen(1)
```

See RESERVE.

CHARCOPY *(Copy a character set into a particular bank)*

CHARCOPY s TO b

The CHARCOPY instruction copies character set *s* to bank number *b*. Values of 1 to 3 correspond to the system sets, and numbers 4 to 16 denote user-defined sets.

Example:

```
reserve as set 5,charlen(1)
```

Reserve bank 5 as set of the same length as system set 1.

```
charcopy 1 to 5
```

Copy system set 1 into bank 5.

See CHARLEN, RESERVE.

Using a character set from a window

1. Find the size of the new set using DIR "*.mbk". Round this up to the nearest 1,000 bytes just to be on the safe side.
2. Reserve some space for the set using RESERVE AS SET.
3. Load your file into this bank with a line like LOAD "filename.mbk",n where *n* is the number of the bank you are using to hold the set.
4. Repeat phases 1 to 3 for each new set.
5. Open a window using WINDOPEN. Set the character set number value to 3 plus the number of your set. Note you can avoid stages 1 to 3 when installing a single character set by choosing the Quit & Grab option from the font definer.

Example:

```
reserve as 5,4000: rem Assumes set is 8x8
load "FONT1.MBK",5: rem Load example font into bank 5
```

Type in the following program. It creates a window, and outputs the entire character set on to it.

```
new
10 windopen 1,1,1,38,23,1,4
20 for I=32 to 255
30 print chr$(i);
40 next I
50 wait key
```

Simple isn't it.

If you like, you can edit this set using the FONTS.ACB accessory. Now for a

somewhat larger example which displays five different character sets on the screen at once.

```
new
dir "*.mbk"
reserve as set 5,5000
load "FONT1.MBK",5
reserve as set 6,5000
load "FONT2.MBK",6

10 rem Multiple character set example.
20 rem Displays 5 character sets on the screen at once
30 rem Mode 1 looks rather better then mode 0.
40 rem Remove line 50 for mono monitors
50 mode 1 : cls
60 for I=1 to 5
70 rem Define windows using WINDOPEN
80 if I<4 then windopen I,(I-1)*26+1,0,26,12,I,I else windopen I,(I-
4)*26+1,12,26,12,I,I
90 rem Output all printable characters in window
100 for J=32 to 255
110 print chr$(J);
120 next J
130 next I
140 goto 140
```

Changing the default sets

When STOS Basic is loaded, it automatically installs three system sets into the ST's memory. These sets are stored in the STOS folder under the following names:

8X8.CR0 (Default set for low resolution)
8X8.CR1 (Default set for medium resolution)
8X16.CR2 (Default set for high resolution)

If you change the contents of these files, you can modify the default character set for your particular resolution and the ST will boot up using your own customised character set.

In order to do this you need to follow the following procedure:

- Create your new set using the FONTS.ACB accessory.
- Load your set into bank 5 of the current program using the Quit & Grab option.
- Place a copy of your system disc into the drive, and type one of the three lines below, depending on the resolution you normally use.

Low resolution bsave "\STOS\8X8.CR0",start(5) to start(5)+length(5)
Medium resolution bsave "\STOS\8X8.CR1",start(5) to start(5)+length(5)
High resolution bsave "\STOS\8X16.CR2",start(5) to start(5)+length(5)

As a demonstration of this technique, load the file FONT1.MBK into the FONT accessory using the Load File option from the Disc menu. Now use the QUIT & GRAB option to return to the editor. Insert your copy of the STOS Basic system disc into the drive. DO NOT USE YOUR ORIGINAL SYSTEM DISC FOR THIS PURPOSE! Type in one of the three lines above to set the default set for any of

the three possible resolutions.

When you reboot the copy of the STOS Basic disc, STOS will now load and use the new font.

Note that STOS Basic can also load up to six supplementary sets as well. These should have the extensions .CR4 to .CR9, and can be accessed using the character set numbers four to nine respectively. Otherwise the method used to save them is identical to that explained above. If some of these extra sets have been loaded, the numbers of any new sets you define need to be incremented accordingly.

Note that the size of these sets is determined when you created them with FONT.ACB. This means you can readily use any of these six supplementary sets for all three graphics modes.

Icons

The STOS Basic Icons are a group of useful 16 by 16 characters, stored in bank number 2. These icons can be output to the screen at the current cursor position using PRINT. This allows you to use them to create complicated backgrounds for your games. You can also incorporate icons directly into a menu. See Chapter 9 ior more details. We've provided a special set of icons especially for your use in the file ICONDEMO.MBK.

ICON$ *(Generate an icon at the current cursor position)*

ICONS(n)

In order to output an icon to the screen you simply print a string containing a CHR$(27) character followed by CHR$(*n*), where *n* is the number of the icon you wish to draw. This string can be generated directly using the ICON$ function.

Example:

```
new
load "ICON.MBK"
10 forX=0 to 19
20 for Y=0 TO 4
30 locate X*2, Y*2
40 print icon$(X*5+Y+1)
50 next Y
60 next X
```

Also:

```
print chr$(27)+chr$(5)
This is equivalent to print icon$(5)
```

The icon definer

This is very similar to the font definer accessory, but rather less involved. It can be loaded using the line:

```
accnew:accload "ICONS.ACB"
```

You can now access this accessory from the editor at any time using Help+f1. On startup you are presented with menu and two windows. The bottom window occupies the entire width of the screen and is used to select an icon to be edited.

9 Menu commands

STOS Basic provides you with a number of clever facilities for creating and manipulating on-screen menus. Although these menus may look rather different to their Gem equivalents, they are considerably more powerful. They are also a great deal easier to use. The best way to explain the commands is by writing a complete program which is developed in this chapter.

Creating a menu

Before you can incorporate one of these menus into a program, you first need to define the menu titles which will be displayed on the screen. This is done with the MENU$ command.

MENU$

MENU$(x)=title$ [,paper,pen]

Title$ holds the title of your menu, and *paper* and *pen* are the colours of each heading and background respectively. The value of *x* denotes the number of the menu whose title you wish to create.

These menus are given numbers from 1 to 10 starting from the left hand corner of the screen. Here is a simple example which constructs a menu consisting of just two titles: ACTION and MOUSE.

```
new
10 menu$ (1)="ACTION "
20 menu$ (2)="MOUSE"
```

You can now specify a list of options to be associated with each of these titles using a second form of the MENU$ command.

MENU$(x,y)

MENU$(x,y)=OPTION$ [paper,pen]

The variables *X* and *Y* in this instruction refer to the title number, and the option number of the menu line. The string *option$* represents the menu text. You can, however, use any string you like for this purpose.

Type the following lines into your program:

```
25 rem Action menu
30 menu$ (1,1)="Quit"
35 rem Mouse menu
40 menu$ (2,1)="Arrow"
50 menu$ (2,2)="Hand"
60 menu$ (2,3)="Clock"
```

This will determine the various alternatives for the ACTION and the MOUSE menus. If you try to run this program as it stands, nothing happens. The reason

9 Menu commands

STOS Basic provides you with a number of clever facilities for creating and manipulating on-screen menus. Although these menus may look rather different to their Gem equivalents, they are considerably more powerful. They are also a great deal easier to use. The best way to explain the commands is by writing a complete program which is developed in this chapter.

Creating a menu

Before you can incorporate one of these menus into a program, you first need to define the menu titles which will be displayed on the screen. This is done with the MENU$ command.

MENU$

MENU$(x)=title$ [,paper,pen]

Title$ holds the title of your menu, and *paper* and *pen* are the colours of each heading and background respectively. The value of *x* denotes the number of the menu whose title you wish to create.

These menus are given numbers from 1 to 10 starting from the left hand corner of the screen. Here is a simple example which constructs a menu consisting of just two titles: ACTION and MOUSE.

```
new
10 menu$ (1)="ACTION "
20 menu$ (2)="MOUSE"
```

You can now specify a list of options to be associated with each of these titles using a second form of the MENU$ command.

MENU$(x,y)

MENU$(x,y)=OPTION$ [paper,pen]

The variables *X* and *Y* in this instruction refer to the title number, and the option number of the menu line. The string *option$* represents the menu text. You can, however, use any string you like for this purpose.

Type the following lines into your program:

```
25 rem Action menu
30 menu$ (1,1)="Quit"
35 rem Mouse menu
40 menu$ (2,1)="Arrow"
50 menu$ (2,2)="Hand"
60 menu$ (2,3)="Clock"
```

This will determine the various alternatives for the ACTION and the MOUSE menus. If you try to run this program as it stands, nothing happens. The reason

for this is that STOS Basic first requires you to use a special command to start your new menu running.

MENU ON

Add the following line to make the program work properly:

```
70 menu on
```

MENU ON has a number of possible extensions. These allow you to choose any one of 16 different borders for your menus. You can also use this function to change the current menu style.

STOS Basic supports two distinct types of menu: Drop-down menus and pull-down menus. Drop-down menus are selected whenever the mouse touches the menu line, whereas pull-down menus also require you to press the left mouse button as well. The full definition of the MENU ON statement is therefore:

MENU ON [border][,mode]

border can range from 1 to 16.

mode is either 1 for a drop-down menu or 2 for a pull-down menu.

If you want to use pull-down menus in your program, you can replace line 70 with:

```
70 MENU ON 5,2
```

This generates a pull-down menu with border type 5. There's also a number of other useful options:

MENU OFF

Permanently switches off the entire menu and clears the menu from the ST's memory.

MENU FREEZE

Temporarily freezes the action of the menu. The menu can be restarted with MENU ON.

MENU$(title,option) OFF

This instruction disables one of the list of menu items under *title*. Any further attempts to call this entry are completely ignored.

MENU$(title,option) ON

Reverses the effect of the above instruction.

STOS stores all your menus in bank number 15. This bank should therefore only be reserved when these menus are not required in your program.

Making a selection

The menu you have prepared is now ready for use. It can be read using the two reserved variables: MNBAR and MNSELECT.

MNBAR and MNSELECT

MNBAR holds a number denoting the menu title you have chosen, while MNSELECT contains the number of the specific option you have highlighted with the mouse. You can see how this works by entering lines 90-110:

```
90 OPTION=mnbar : CHOICE=mnselect
100 print "Title Number ";OPTION; " Selection     Number";
CHOICE
110 goto 90
```

If you run this program, the title number and the option number you have selected will be displayed to the screen.

This code can be expanded into a real program, by replacing the lines 100 onwards with:

```
100 if OPTION=1 and CHOICE=1 then menu off : stop
110 if OPTION=2 and CHOICE<>0 then change mouse CHOICE
120 goto 90
```

Line 100 tests the menu to see if you have decided to exit from the program. The action of line 110 is to check whether you wish to swap the mouse pointer. It can then use this information to alter the pointer type with a CHANGE MOUSE instruction.

ON MENU

The last example was fairly simple. But supposing you wanted to write a routine with a larger and more complicated series of menus. In this case, your program would need to use a long list of IF...THEN statements to deal with each and every possibility. Inevitably this would make your program both unwieldy and hard to change. It would therefore be better if there was an easier way of handling these menus.

Fortunately STOS Basic includes a special ON MENU statement which provides you with a painless method of managing even the largest menus. It does this by automatically jumping to one of a list of line numbers, depending on the title you have chosen.

ON MENU GOTO line1 [,line2]...
is broadly equivalent to the line:
ON MNBAR GOTO line1[,line2]...

One major difference between the above instruction and ON MENU is that ON MENU is performed using interrupts. This allows your program to execute another task at the same time as your menus are being tested.

Example:

```
new
10 T=0
20 menu$ (1)=" ACTION"
30 menu$ (1,1)="COUNT"
40 menu$ (1,2)="QUIT"
50 menu on
60 on mnbar goto 90
80 T=T+1 : goto 80
90 if mnselect=1 then locate 0,1 : print T : goto 60
```

```
100 if mnselect=2 then stop
```

When you run this program, it first creates a menu, and then checks whether this menu has been accessed. It now reaches line 80 and repeatedly adds 1 to the variable *T*. Since line 60 is never executed again, playing around with the menu has no effect whatsoever. Try replacing line 60 with:

```
60 on menu goto 90
70 on menu on
```

In this case the menu will function perfectly, despite the fact that the program is still stuck at line 80. Furthermore, every time you choose COUNT, you will find that the value of the variable T has increased.

This appears to prove that line 80 is running at the same time as line 60. What is really happening is that the menus are being tested by STOS Basic 50 times a second using an interrupt similar to that utilised by the sprite commands.

The entire process is set in motion by the ON MENU ON instruction. As you might expect, there's also a ON MENU OFF command which turns the menus off. You can use this on menu routine in conjunction with any sequence of Basic instructions you like, providing they make no attempt to input or output information to the screen.

Up until now the examples have been fairly trivial. We will therefore go on to describe how a STOS Basic menu can be incorporated into a real program. To that end, we'll produce a small, but useful version of Doodle, directly comparable to that found on the ST startup disc. As before, we will begin by defining the menu:

```
new
3 mode 0
5 rem Action menu
10 menu$ (1)=" ACTION "
20 menu$ (1,1)="DRAW"
30 menu$ (1,2)="QUIT"
35 rem Pen menu
40 menu$ (2)=" PENS "
50 menu$ (2,1)="Small"
60 menu$ (2,2)="Medium"
70 menu$ (2,3)="Large"
75 rem Colour menu
80 menu$ (3)=" COLOUR "
90 for I=1 to 16
100 menu$ (3,I)="<six spaces>",I-1,0
110 next I
```

At first glance lines 90 to 110 seem to produce a menu consisting of nothing more than blank spaces. But if you look more closely you'll see that we're actually setting the paper colour of each line to the value of I-1. This neatly turns our spaces into a bar of the appropriate colour – a technique which is used to great effect by many of the accessories on the disc.

Note that in order to keep things as simple as possible, we've assumed that the maximum number of colours available is 16. People with mono monitors should therefore delete line 3 and alter line 90 to:

```
90 for I=1 to 2
```

You must now activate the menu using the MENU ON command.

```
120 menu on
```

Before you can continue, you need to decide precisely where the program should go when each of the menu titles are selected. In this example we've placed the routines starting at 200, 400 and 600 respectively.

```
150 on menu goto 200,400,600
160 on menu on
170 goto 170
```

When a menu item is chosen, line 150 will automatically execute the routines at either 200, 400 or 600 depending on whether the titles ACTION, PEN or COLOUR were picked. Incidentally the reason for the line at 170 is to give STOS Basic something to do while the program is waiting for the menu to be used.

We'll now examine the ACTION routine at lines 200-400 which effectively forms the heart of the Doodle program. ACTION gives you a choice between two different alternatives: Exit or Draw. If you select the Exit option then the program should simply return to the editor.

```
199 rem Actions
240 M=mnselect
250 if M=2 then menu off : stop
```

The second possibility is that you might wish to actually do some drawing on the screen. It's easy enough to detect whether this feature has been chosen using a simple IF...THEN statement.

```
260 rem If item 1 not picked go back to menu loop
270 if M<>1 then 150
```

Now comes the drawing routine itself which is rather more complicated. We will begin by specifying precisely what we want the program to do and then see how this effect will be achieved. What we require is a small routine to input the position of the mouse, and then draw a filled circle at the appropriate coordinates whenever the left mouse button is pressed. In order to enable the user to draw continuous lines, this process should be repeated until the drawing routine is terminated with the right button.

```
280 rem Draw until right mouse button clicked
290 repeat
300 rem Wait until a mouse button has been pressed
310 repeat: M=mouse key : until M<>0
320 rem If left button then draw a circle of radius SIZE*5
330 if M=1 then X=x mouse : Y=y mouse : circle X,Y,SIZE*5
340 until mouse key=2: rem Check for right mouse
390 goto 150
```

The code to deal with the other two menu items is very simple indeed since it only has to read the menu using mnselect and then use this to set either the size or the colour of the pen.

```
399 rem SIZE = size of pen
400 SIZE=mnselect : goto 150
599 rem C = Colour of pen
600 C=mnselect : if C>0 then ink C-1
610 goto 150
```

The initial value for SIZE needs to be set to one. There also needs to be another line to prevent a flashing text cursor in the top left hand corner of the screen.

```
85 size=1
130 curs off : clw : rem Get rid of the flashing cursor and clear screen
```

Another problem is that the drawing operations can occasionally clash with the menu. In extreme cases this can lead to almost total destruction of the menu line itself. There are two things that can be done to avoid this difficulty. Firstly you can turn off the menus during the drawing operations using MENU FREEZE.

As an additional safeguard, it's also a good idea to restrict the mouse to the part of the screen below the menus with the LIMIT MOUSE command. This stops you from accidentally obliterating large sections of the menu line with part of your drawing.

```
200 menu freeze : rem Switch off menu
210 rem Limit mouse to below menu. Modify for use in high or medium res
220 limit mouse 0,22 to 300,180
350 menu on : rem Restart menu
360 limit mouse : rem Remove mouse limit
```

Finally, the mouse pointer has a completely different effect depending on whether you are drawing a circle or calling one of the menus. We therefore changed the mouse pointer to a hand within the drawing routine, to avoid any possibility of confusion.

```
230 change mouse 2 : rem Change mouse to hand
370 change mouse 1 : rem Change mouse back to arrow
```

Icons

So far, all the menus we have created have been composed of text. However you can also incorporate icons into a menu:

MENU$(1)=ICON$(2) *Loads the title number with icon two.*

MENU$(2,1)=ICON$(3) *Associates icon 3 with option 1 of title 2.*

To demonstrate how this works, there are some icons for the Doodle program in the file ICON.MBK. This should first be loaded from the editor using LOAD "ICON.MBK".

You should now replace lines 50 to 70 with:

```
50 menu$ (2,1)=icon$(3):rem Small circle
60 menu$ (2,2)=icon$(2):rem Medium-sized circle
70 menu$ (2,3)=icon$(1):rem Large circle
```

These lines substitute the original PEN menu with a set of three icons representing the various possible pen sizes. When you execute this program, these icons can be accessed with the mouse in exactly the same way as a normal menu.

Possible ideas for expansion

The previous example could form the basis of quite a powerful drawing utility. Here are a few of the possible ways you could expand it.

1. Add a Disc menu to allow the loading and saving of pictures via the disc. (Use something like LOAD F$+".NEO" or SAVE F$+".NEO" where *F$* is the name of your file).

2. Improve the resolution of your picture by using points instead of circles.
3. Add an eraser.
4. Replace the hand pointer with cross-hairs. This can be achieved by using the Sprite Editor program to generate a sprite of the appropriate shape, and then calling *change mouse* using the image number plus 4.
5. Add routines to draw other objects such as boxes or ellipses.
6. Implement a cut and paste feature using SCREEN COPY.
7. Change the size of parts of the picture using ZOOM or REDUCE.

Troubleshooting

As you have seen, using menus from STOS Basic is normally very easy indeed. Even the best of us however, can occasionally make a mistake, and when this happens it may help to check the following list of common problems.

Problem: The Menu flickers and dies every time you try to call it with the mouse.
Solution: You have ordered a menu out of sequence. Check the menu definitions.

Problem: The menu doesn't appear in your program.
Solution: You may have forgotten to use the MENU ON command.

Problem: ON MENU doesn't work.
Solution: Check whether there is an ON MENU ON statement. Also make sure the program isn't attempting to perform Input or Output to the screen while ON MENU is active.

10 Other commands

Up until now we have concerned ourselves with many of the more exciting features of STOS Basic. But like all versions of the Basic language, STOS also includes a variety of more mundane facilities which allow you to do a range of useful things such as accessing the ST's screen, keyboard or disc.

The aim of this chapter is therefore to provide you with all the information you need to familiarise yourself with the nuts and bolts of the STOS Basic system. Whenever possible. We have included any major differences between STOS and standard Basic. This should make it fairly easy to convert programs written in most other dialects of Basic for use with this package. Since the scope of this manual cannot extend to providing an in-depth tutorial on Basic itself, we have provided a number of worked examples which should prove useful even for a complete beginner.

Control Structures

GOTO *(Jump to a new line number)*

GOTO is probably the most commonly used of all the Basic instructions. The action of a GOTO is to transfer the control of the program from the current line number, to a new one.

GOTO line number	Where *line number* can be any line in your Basic program.
GOTO expression	*expression* can be any allowable STOS Basic expression involving either variables or constants. Technically this is known as a computed goto.

Example:

```
new
10 goto 30
20 print "This line is never printed"
30 print "Now executing line 30"
```

Now for an example of a computed GOTO.

```
new
10 JUMP=10
20 goto JUMP*2+20 : rem same as goto 40
30 print "This line is never printed"
40 print "Jumped to line ",JUMP*2+20
```

This example is really a rather bad piece of programming, because any mistake you make in line 10 or 20, could lead to your program jumping somewhere totally unforseen. Furthermore, these computed gotos are invariably far slower than normal ones, and make it almost impossible to renumber your program. They should therefore be used with extreme caution.

Users of other Basics should note that STOS Basic does not support any form

of labels. This means that you should remember to place a number at the start of each and every line. See AUTO

If you absolutely have to use labels in your program, you can simulate them with a computed goto like so:

```
100 LABEL=120
100 goto LABEL
110 goto 110
120 print "Label reached"
```

Finally, GOTOs should NEVER be used to jump inside a FOR...NEXT loop, as this will lead to a *NEXT WITHOUT FOR* error.

See also ON GOTO

GOSUB *(Jump to a Subroutine)*

This is very similar to GOTO, but has the additional bonus of enabling you to jump back where you started with a RETURN instruction. The most common use of GOSUB is to allow you to split a program into smaller, more manageable chunks, known as subroutines. As with GOTO, there are two different forms of the GOSUB instruction.

GOSUB line	Jump to the subroutine at *line*.
GOSUB expression	Jump to the subroutine at the number given by the result of *expression*

Example:

```
new
10 I=1
20 gosub 40
30 goto 20
40 print "You have called this gosub ";I;"times"
50 inc I
60 return
```

This demonstration was trivial, but if you have a look at some of the programs on the disc, you will find many real examples of just this sort of subroutine.

RETURN *(Return from a GOSUB to the next instruction)*

RETURN exits from a subroutine, and jumps back to the statement after the initial GOSUB.

Example:

```
new
10 gosub 100:print "Returned"
20 end
100 print "Inside Gosub":return
```

POP *(Remove the RETURN information after a GOSUB)*

The POP instruction removes the return address generated by a GOSUB and allows you to leave the subroutine without having to execute the final RETURN statement.

Here is an example of this instruction in action:

```
new
10 I=1
20 gosub 40
30 goto 20
40 print "You have called this gosub ";I;"times"
50 inc I : if I>100 then pop:goto 70
60 return
70 print "Gosub terminated after ";I-1;" Times"
```

See ON GOSUB

FOR...NEXT *(Repeat a section of code a specific number of times)*

This is the classic way of repeating parts of a Basic program. The format of the instruction is:

FOR var=start TO finish [STEP inc]

list of instructions

NEXT [var]

When this loop is first entered, *var* is loaded with the value of *start*. The instructions between the FOR and the NEXT are now performed until the NEXT is reached. The NEXT instruction increments var by either *inc*, or 1, depending on whether the optional STEP has been included. The loop counter is now tested. If *var* is either greater than *finish* (for positive increments), or less than *finish* (for negative steps), the loop is terminated, and the instruction after the NEXT is executed. Otherwise the loop is restarted from the top.

Here are a couple of examples of FOR...NEXT loops.

```
for 9=1 to 100 step 10:print 9:next 9
```

```
new
10 for a=32 to 255
20 print chr$(a);
30 next a
```

```
new
10 for R1=20 to 100 step 20
20 for R2=20 to 100 step 20
30 for a=0 to 3
40 ink a
50 ellipse 160,100,R1,R2
60 next a
70 next R2
80 next R1
```

See how we've placed a number of FOR...NEXT loops inside each other. This is known as nesting. STOS Basic will permit you to nest anything up to a maximum of 10 FOR...NEXTs in this way. Unlike some other Basics, STOS Basic does not allow you to replace lines 50-70 with "NEXT I,R1,R2". All NEXT instructions should be placed directly at the correct point in the program.

WHILE...WEND *(Repeat a section of code while a condition is true)*

This instruction enables you to repeat a series of instructions until a specific condition has been satisfied.

WHILE condition

list of statements

WEND

The *condition* can be any set of tests you like, and can include the constructions AND and OR. This check is always performed at the start of the WHILE loop. The *list of statements* between the WHILE and the WEND will be only be executed if this condition is true.

Type the following example:

```
new
10 input "Type in a number";X
20 print "Counting to 11"
30 while X<11
40 inc X
50 print X
60 wend
70 print "Loop terminated"
```

The number of times the WHILE loop in this program will be executed depends on the value you input to the routine. If you type in a number larger than 10, you will find that the loop is not entered at all.

As a rule, these WHILE loops should therefore only be used when a list of statements needs to be repeated 0 or more times. The program above is effectively equivalent to the following routine written in standard Basic:

```
new
10 input "Type in a number";X
20 print "Counting to 11"
30 if X>=11 then 70
40 inc X
50 print X
60 goto 30
70 print "Loop terminated"
```

It should be readily apparent that the program with the WHILE statement is much easier to read than the one which used GOTO. Each WHILE instruction in your program should be matched by exactly one WEND statement. See REPEAT...UNTIL

REPEAT...UNTIL *(Repeat a section of code until a condition is satisfied)*

This pair of statements is similar to WHILE...WEND except that the test for completion is made at the end of the loop rather than the beginning. Furthermore, the action of the UNTIL statement is to continue executing the loop until the condition is FALSE. The format of this instruction is:

REPEAT

list of statements

UNTIL condition

where condition is a list of conditions, and the *list of statements* can be any set of Basic instructions you like.

Here is a small example, taken from the Doodle program in Chapter 9:

```
10 repeat
20 M=mouse key : rem test to see if mouse button pressed
30 until M<>0
40 print "You clicked on the mouse button"
```

we could have used a WHILE...WEND construct in this program instead. This would have changed the routine to:

```
10 M=mouse key
20 while M=0
30 M=mouse key
40 wend
50 print "You clicked on the mouse button"
```

In this case, we would have had to use an extra instruction to test for the mouse key at the start of the loop.

Since a REPEAT...UNTIL loop always executes at least once, this was not needed in the first example. As with WHILE...WEND, you should always remember to match each REPEAT with an UNTIL.

STOP *(Stop running the program and return to the Editor)*

This command stops the current program running and returns to the editor. It can be used at any point in your program.

Example:

```
new
10 input "Input a number between 1 and 100 (0 to stop)";N
20 if N=0 then stop
30 for I=1 to N
40 print I*I
50 next I
60 goto 10
```

Note that unlike END, a program terminated with STOP can be restarted with CONT, providing it has not been altered in the meantime using the editor.

END *(Exit from the program)*

This instruction exits from a program and returns to the editor. Programs which have been terminated using END cannot be subsequently restarted using CONT.

See STOP.

IF ... THEN [ELSE] *(Choose between alternative actions)*

The IF...THEN instructions allow you to make decisions within a Basic program. The format is:

IF conditions THEN statements1 [ELSE statements2]

conditions can be any list of tests including AND and OR.

Statements1 and *statements2* can be either lists of STOS Basic instructions, or line numbers.

The action of the IF...THEN instruction is to execute the instructions in *statements1* if the *conditions* are true. If the optional ELSE statement is included, then *statements2* will be performed when the *condition* is false. Otherwise control will pass to the line after the IF...THEN instruction. The following example program demonstrates most of the various possibilities.

```
10 input "Input a number";N
20 print "Number ";N;" is ";
30 if N>0 then print "Positive"; else print "Negative";
40 if (N/2)*2=N then print " and Even" : goto 60
50 if (N/2)*2<>N then print " and Odd"
60 input "Continue Y or N";A$
70 if A$<>"Y" and A$<>"y" then 90 else 10
80 print "Never executed"
90 stop
```

Note that STOS Basic restricts these IF...THEN statements to a single line. See NOT,TRUE,FALSE

ON...GOTO *(Jump to one of a list of lines depending on a variable)*

ON var GOTO line1,line2,line3...

The ON GOTO instruction allows your program to jump to one of a number of lines depending on the value of the variable *var*. If *var* takes a value of 1, for instance, the instruction is identical to a simple GOTO *line*1. Similarly, if var holds a 2 then the program will branch to *line2*, and so on. In order to have an effect, the ON...GOTO statement requires *var* to hold a figure between 1 and the number of possible destinations. Look at the following small example:

```
new
10 input "Input a number ";N
20 on N goto 50,60,70,80
30 print "You input a number either less than 0 or greater than 4"
40 goto 10
50 print "You input the number ONE" : goto 10
60 print "You input the number TWO" : goto 10
70 print "You input the number THREE" : goto 10
80 print "You input the number FOUR" : goto 10
```

Note that the variable used for N must always be an integer.

See GOTO, GOSUB, ON GOSUB

ON...GOSUB *(GOSUB one of a list of routines depending on a var)*

ON var GOSUB line1,line2,line3...

This is identical to ON...GOTO except that it uses a gosub rather than a goto to jump to the line. When the subroutine has finished executing, it should use a RETURN to jump back to the next instruction after the ON...GOSUB statement.

Example:

```
new
10 input "Input a number ";N
20 on N gosub 50,60,70
40 goto 10
50 print "Subroutine ONE" : return
60 print "Subroutine TWO" : return
70 print "Subroutine THREE" : return
```

See also GOSUB and ON GOTO

ON ERROR GOTO *(Trap an ERROR within a Basic program)*

This command is used to allow the detection and correction of errors which occur within a STOS Basic program. Take, for instance, the following routine:

```
10 input "Input a positive number";N
20 print "The Square Root of ";N;" is ";SQR(n)
30 goto 10
```

This program works fine until you try to type in a negative number. When this happens an error is generated, as you are not allowed to calculate the square root of any number less than 1. STOS Basic therefore returns you to the editor, and prints out the error message *ILLEGAL NEGATIVE OPERAND in line 20.*

You can avoid this problem by trapping the error with an ON ERROR GOTO instruction. The format is:

ON ERROR GOTO line

Where *line* is the location of your new error correction routine.

line refers to the location of a routine which will be executed whenever an error occurs. You can also use an expression for this purpose, but this is generally rather a bad idea as the expression is only evaluated once, when the ON ERROR GOTO instruction is first initialised.

Example:

```
10 on error goto 50
20 input "Input a positive number";N
30 print "The square root of ";N;" is ";sqr(N)
40 goto 10
50 print
60 print "I'm afraid you can only take the square root of a
positive number"
70 N=abs(N)
80 resume 10
```

In order to turn the action of ON ERROR GOTO off, you simply type the line: ON ERROR GOTO 0

See RESUME, ERRN, ERRL, ERROR

RESUME *(Resume execution of the program after an error)*

This instruction is used from within an error trap created by ON ERROR GOTO.

The action of RESUME is to jump back to the part of the program which caused the problem, after the error has been corrected by your routine. You should NEVER attempt to use GOTO in this context.

RESUME has three possible formats:

RESUME	Jump back to the statement which caused the error and try again.
RESUME NEXT	Jump to statement following the one which generated the error.
RESUME *line*	*line* Jump to line number.

See ON ERROR GOTO, ERROR, ERRL, ERRN

ERRN *(Reserved variable containing the number of the last error)*

When an error occurs, ERRN is automatically loaded with the error number. This can be printed out using a line such as:

PRINT ERRN

ERRL *(Reserved variable holding the location of last error)*

ERRL contains the line number of the last error which occurred.

Here is a small example.

```
10 rem Error test routine
20 on error goto 50
30 rim I appear to have made a slight mistake!
40 stop
50 print "ERROR NUMBER ";errn;" at line ";errl
60 resume next
```

See also ERRN, ERROR and ON ERROR GOTO

ERROR *(Generate an ERROR and return to the STOS Editor)*

The action of the ERROR command is to actually generate an error. This may sound rather crazy, but it's often quite useful. Supposing you have created a nice little error handling routine which is able to cope with any possible disc errors.

```
error 2
```

Quits the program and prints out an out of memory error.

The most common form of this instruction is:

```
error errn
```

This uses the ERRN function to print the current error condition.

By testing the ERRN for the errors your program can correct, you only need to revert back to the editor when absolutely necessary.

BREAK *(Turn on or off the Control+C Break key)*

Normally you can interrupt a program and return to the editor at any time by pressing the two keys Control and C. Although this is useful when you're debugging a program, it would be very dangerous to allow this function to operate in a commercial games program, as it would make it extremely easy for an unscrupulous person to steal some of your code. You can therefore turn this function off using a special BREAK OFF command.

As you might expect, you can also reactivate the Break keys using:

```
break on
```

But be warned: NEVER run a protected program unless you have made a backup copy on the disc first. Otherwise if the program gets stuck in a loop, you could easily end up losing several hours of your work.

The keyboard

KEY *(Function to assign a string to a function key)*

Any of the 10 function keys can be assigned a string of up to 64 characters long using the KEY command.

KEY(x)=a$

Assigns string *a$* to key number X.

a$ is the string which will be returned whenever key X is pressed. X is a number from 1 to 20, where the numbers between 11-20 represent a shifted version of the normal function keys.

Example:

```
1 rem Reassign function keys. Warning! In order to get the
2 rem default assignments back, you will need to reboot STOS Basic!
10 for I=1 to 20
20 read A$
30 key (I)=A$+""
40 next I
50 input "Press a function key";F$
60 print "Function key number ";F$
70 goto 50
80 data "one","two","three","four","five","six","seven","eight","nine"
90 data "ten","eleven","twelve","thirteen","fourteen","fifteen","sixteen"
100 data "seventeen","eighteen","nineteen","twenty"
```

If you now run this program, and press a function key, the number of the key you pressed will be printed on the screen.

See also KEY LIST and FKEY

INKEY$ *(Function to get a keypress)*

The INKEY$ function allows you to test whether a key has been pressed at any time, without having to interrupt the action of the program. INKEY$ is used in the

following way.

K$=INKEY$

where *K$* is the string variable which will be used to hold the key which has been pressed.

If the user presses a key, then *K$* will contain the Ascii character which has been input, otherwise *K$* will be set to the empty string "". Ascii values range from 0-255 and represent a standard code used to hold all alphanumeric characters. It is important to note that some keys, such as the cursor keys, and the function keys, use a rather different format. These must therefore be read using a separate SCANCODE function.

Example:

```
new
10 while K$=""
20 K$=inkey$
30 wend
40 print "You pressed the ";K$;" Key with an
Ascii code of ";asc(K$)
50 K$="" : goto 10
```

See CLEAR KEY and SCANCODE

SCANCODE *(Input the SCAN CODE of the last key input with INKEY$)*

SCANCODE is used in conjunction with INKEY$ to test whether the user has pressed a key which does not return an Ascii code. If INKEY$ detects that such a key has been input, it returns a character with the value 0. When this happens you should use the SCANCODE function to determine the internal code associated with this key.

Try typing in the following small example:

```
new
10 while K$=""
20 K$=inkey$
30 wend
40 if asc(K$)=0 then print "You Pressed a key with no ASCII code."
50 print "The scancode is";scancode
60 K$="" : goto 10
```

CLEAR KEY *(Initialise keyboard buffer)*

Whenever you type a character on the ST's keyboard, its Ascii code is placed in an area of memory known as the keyboard buffer. It is this buffer that is read by the INKEY$ function. At the start of a program the buffer may well be full of unwanted information. It's therefore generally a good idea to remove all this garbage first using CLEAR KEY.

Add line 5 to the program in the previous example.

```
5 clear key
```

See PUT KEY,INKEY$

INPUT$(n) *(Function to input n characters into a string)*

INPUT$ reads *n* characters from the keyboard, waiting for each one, and then loads them into a string. As with INKEY$, these characters are not echoed back on the screen.

X$=INPUT$(n)

X$ represents any string variable and *n* is a number denoting the length of the string to be input.

Example:

```
new
10 clear key
20 print "Type in ten characters"
30 C$=input$(10)
40 print "You typed in the string ";C$
```

It is important not to confuse INPUT$ with INPUT, as the two instructions are completely different.

Also note that there is a special version of INPUT$ which is used to access the disc.

FKEY *(Read the function keys directly)*

FKEY is a special form of the INKEY$ function which can be used to test the function keys directly without having to tediously use SCANCODE. Whenever a function key is pressed, FKEY returns a number between 1 and 20. Numbers greater than 10 indicate that the key has been shifted, and a value of zero means that no key has been pressed.

FKEY is often used in conjunction with ON...GOSUB to jump to one of a number of subroutines depending a function key chosen by the user.

ON FKEY GOSUB line1,line2,line3...

See KEY, KEY LIST

WAIT KEY *(Wait for a keypress)*

The action of WAIT KEY is simply to halt the program until the user hits a key.

Example:

```
new
10 print "Press a key"
20 wait key
30 print "Key pressed"
```

KEY SPEED *(Change key repeat speed)*

KEYSPEED repeatspeed, delay

This instruction allows you to tailor the speed of the keyboard to your own particular taste. *repeatspeed* is the delay in 50ths of second between each repeated

character. *Delay* is the time in 50ths of a second between pressing a key, and the start of the repeat sequence.

PUT KEY *(Put a string into the keyboard buffer)*

This function is used to load a string of characters into the keyboard buffer. Carriage returns can be included in this string using the ' character. The most common use of PUT KEY is to call up a direct mode command after a program has terminated.

Example:

```
10 put key "new'"
```

When this line is executed, the program erases itself from the ST's memory. It does this by placing a "new" into the keyboard buffer, which is then performed directly from the editor when the program ends.

Input/output

INPUT *(Input a number or some text into a string variable)*

INPUT provides you with a standard way of inputting information into a variable. There are two possible formats for the instruction:

INPUT variable list	*variable list* can be any list of variables separated by commas.
INPUT "Prompt";variable list	*Prompt* may be any string of characters you like.

When you execute an INPUT instruction, the ST displays a ? and waits for you to enter the required information from the keyboard. If an optional prompt has been included, then this will be printed out instead of the "?".

Example:

```
new
10 input A
20 print A
```

If you now run this program and type in the number 10, the following dialogue will ensue. In order to distinguish between your input, and the computers output, We've underlined anything entered from the keyboard.

run
?10
10

If more than one variable has been specified in the list, these should be entered as in the example below.

```
new
10 input A,B,C$
20 print A,B,C$
```

We'll now show you some sample dialogue of this program in action.

run
? 15,40,string of characters
15 40 string of characters

Notice how we've separated the three values typed in with a comma. Any commas input as part of a string will therefore effectively split the string in two. In some circumstances this might be a major inconvenience, so STOS Basic includes a useful LINE INPUT instruction which allows you to use a Return instead of a comma as the separator.

Here's another example, showing the action of the prompt:

```
new
10 input "Enter your age:";A
20 input "Enter the month, and the year of your birth:";MS,Y
30 input "Enter your christian name and surname:";C$,S$
40 print "Age = ";A
50 print "Month = ";MS;" Year = ";Y
60 print "Name = ";C$,S$
```

```
run
Enter your age:26
Enter the month, and the year of your birth:July,1961
Enter your christian name and surname:Stephen,Hill
Age = 26
Month = July Year = 1961
Name = Stephen Hill
```

Incidentally, if you're used to another version of Basic, you should note that the ; between the prompt and the variables, cannot be replaced by a ,. See INPUT# and LINE INPUT

LINE INPUT *(Input a list of variables separated by a Return)*

Line input is exactly the same as INPUT, except that it uses a Return instead of a comma to separate each variable you type in.

Example:

```
new
10 line input A,B,C$
20 print A,B,C$

run
? 10
?? 20
?? Hello
10 20 Hello
```

See INPUT, LINE INPUT#

PRINT and ? *(Print a list of variables of the screen)*

The PRINT instruction has precisely the opposite effect as INPUT, and prints the contents of a list of variables at the current cursor position on the ST's screen.

PRINT list of variables

The list of variables can include any mixture of strings or numbers. These variables are separated by either a ; or a ,. If a semi-colon ; is used, then the data will be printed immediately after the last variable you output using print. If, however, a

comma is used, the cursor will be positioned a number of spaces ahead. Normally the cursor is moved downwards one line every time a print instruction is executed. This line can be suppressed by placing either of the separators at the end of the PRINT. Note that PRINT can be abbreviated to a ?. This will be expanded in full in any program listings.

Example:

```
new
10 print "This is the story of the Hitchikers Guide to the Galaxy"
20 A=10 : B=20 : C$="Thirty"
30 print A,B;C$
40 print 10,20*10,"Hel";
50 print "lo"
```

See also USING, LPRINT and PRINT#

USING *(Formatted output)*

The USING statement is used in conjunction with PRINT to provide fine control over the format of any printed output.

USING takes a special format string. Any normal alphanumeric characters in this format string will be simply printed out, but if you include one of the characters ~#+-.;^ then one of several useful formatting operations will be performed.

PRINT USING format$;variable list

Note the semi-colon between the format string *format$* and the list of variables.

~ (Shift+#) This is used to format strings. Any occurrences of the ~ are replaced by a character from the following string.

Example:

```
new
10 print using "This is a ~~~~~ demonstration of USING";"Small"
20 print using "1st Letter:~ 2nd Letter:~ 3rd Letter:~";"Basic"
```

If you now type:

```
run
```

these lines will be displayed on the screen.

This is a small demonstration of USING 1st Letter:B 2nd Letter:a 3rd Letter:s

Specifies the number of digits to be printed out from a numeric variable. If this number is greater than the size of the variable then excess # characters will be replaced by spaces.

Example:

```
new
10 print using "####";314211
20 print using "# # # # # #";123456
30 print using "####";56
```

When you run this program it will print out the following lines on the ST's screen.

```
4211
1 2 3 4 5
  56
```

+ This adds a plus sign to a number if it is positive, and a minus sign if it is negative.

Example:

```
new
10 print using "+##";10
20 print using "+##";-10
run
```

displays:

```
+10
-10
```

- This only includes a sign if the number is negative. Positive numbers are preceded by a space.

Example:

```
new
10 print using "-##";10
20 print using "-##";-10
run
```

displays:

```
 10
-10
```

. Places a decimal point in the number, and centres it.

Example:

```
print using "PI is #.###";3.1415926
PI is 3.141
```

; Centres a number but doesn't output a decimal point.

Example:

```
print using "PI is #;###";3.1415926
PI is 3 141
```

^ (Shift+6) Prints out a number in exponential form.

Example:

```
PRINT USING " Here is a number ^";12345.678
```

Here is a number 1.23345678E5

See also FIX

Disc access: sequential files

The Atari ST supports two different types of disc files: Sequential files and random access files.

Sequential files are designed to be used for accessing long lists of information at a time. These files only allow you to read information back from the disc in the precise order it was written. This means that if you want to change just one piece of the data in the middle of the file, you would need to read in the whole file up to and including this value, and then write the entire file back to the disc. STOS Basic allows you to access sequential files for either writing, or reading, but never for both at the same time.

Before you can use one of these files, you first need to open a channel to the file, using OPEN IN or OPEN OUT. You can think of one of these channels as a pipe running from the ST's memory to the file. This pipe is created whenever you open the channel, and can be used to transfer information to and from a disc file, using the INPUT#, or PRINT# instructions respectively. Look at the following small example.

```
new
10 open out #1,"file.seq"
20 input "What is your name";N$
30 print #1,N$
40 close #1
```

This creates a file called FILE.SEQ containing your name. In order to read this information back from the file, type in the lines:

```
new
1 open in #1,"file.seq"
2 input #1,N$
3 print "I remember your name. It is ";N$
4 close #1
```

Notice how both these programs perform three separate operations.

- Open the file using either OPEN IN or OPEN OUT
- Access the file with INPUT#, or PRINT#
- Close the file with CLOSE. Note that if you forget to do this, any changes to the file will be lost!

These three steps need to be completed in exactly this order, every time you access a sequential file. Now for a somewhat larger example.

```
new
10 rem Choose between reading and writing routines
20 input "Do you want to read a file <R>, write a file <W> or stop <RETURN>";A$
30 if A$="R" or A$="r" then 190
40 rem If the user simply press Return then exit
50 if A$="" then stop
60 rem OPEN file "BIRTHDAY.SEQ" for output
70 open out #1,"birthday.seq"
80 rem Input a name and a birthday
90 input "Input the name of your friend or  to stop";F$
100 rem if name =  close file and jump to main routine
110 if F$="" then close #1 : goto 20
120 print F$;"'s Birthday is" : input B$
130 Rem Separate items by a comma for use with INPUT#
```

```
140 print #1,F$;",";B$
150 rem Get another birthday
160 goto 80
170 rem Reading routine
180 rem Dimension strings for WHOLE file. Assumes maximum of 100 bithdays
190 open in #1,"birthday.seq"
200 rem open file for reading
210 dim F$(100),B$(100)
220 rem set item number to zero
230 I=0
240 rem read file until end
250 print "List of birthdays"
260 print "=========================="
270 repeat
280 rem read birthdays
290 input #1,F$(I),B$(I)
300 inc I
310 until eof(1)
320 rem print birthdays
330 for J=0 to I-1
340 print F$(J),B$(J)
350 next J
360 rem close file and go back to start
370 close #1
380 goto 20
```

This program creates a small database consisting of a list of the names and birthdays of your friends. The first half of the routine loads the information into the file BIRTHDAY.SEQ. If this file already exists on the disc, it is erased. You are then prompted to input a list of names and birthdays which are stored on the disc.

The second part of the program opens this file, reads its contents, and displays them on the screen. For more information on sequential files see OPEN IN, OPEN OUT, CLOSE, INPUT#, PRINT#, LINE INPUT#, INPUT$(#Channel,n), LOF, POF, EOF

Disc access: random access files

Random access files are so called because you can access the information stored on the disc in any random order you like. In order to use these files you first need to understand a little bit of theory.

All random access files are composed of units called records, each with their own unique number. These records are in turn split up into a number of separate fields. Every field contains one individual piece of information. When you use sequential files, these fields can be any length you wish, as the file will only be read in one direction. Random access files, however, always require you to specify the maximum size of each of these fields in advance.

Supposing you wanted to produce a file containing a list of names and telephone numbers. In this case you could use the fields:

Field	Maximum length
SURNAME$	15
NAME$	15
CODE$	10
TEL$	10

You could now define these fields using a line like:

```
field #1,15 as SURNAME$,15 as NAME$,10
as CODE$,10 as TEL$
```

It's important to realise that the strings specified by the FIELD instruction can also be used as normal string variables. This allows you to read and write information to any particular field. For example:

```
SURNAME$="HILL" :rem Loads the surname into the field SURNAME$.

TEST$=SURNAME$:PRINT TEST$
```

After you've loaded your record with information, you can write it onto the disc using the PUT command.

Example:

```
put #1,10
```

Loads data into record 10 of file opened on channel 1.

Similarly, you can read a record using the GET instruction.

```
get #1,10
```

Example:

```
10 rem Open file "NAMES.RAN" for random access
20 open #1,"R","names.ran"
30 rem Assign field strings
40 field #1,15 as SURNAME$,15 as NAME$,10 as
AREA$,10 as TEL$
50 rem Choose between reading and writing
60 input "Do you want to read a number <R>, write a number <W>, or exit
<Return>";A$
70 rem exit program if <RETURN> entered. Close file first!
80 if A$="" then close #1 : end
90 if A$<>"W" and A$<>"w" and A$<>"R" and
A$<>"r" then 60
100 rem Get number of record
110 input "Record Number ?";N
120 rem Exit if negative number entered
130 if N<0 then 60
140 if A$="R" or A$="r" then 270
150 rem Routine to write telephone numbers
160 rem Load fields into new record
170 input "Enter the surname";SURNAME$
180 input "Enter the Christian name";NAME$
190 input "Enter the area code ?";AREA$
200 input "Enter the telephone number ?";TEL$
210 rem Store record at position N on disc
220 put #1,N
230 rem Goto main routine
240 goto 60
250 rem Reading routine
260 rem Read record at N into fields
270 get #1,N
280 rem Print fields
```

```
290 print "Record number ";N
300 print "============================
==================="
310 print "Name:";NAME$,SURNAME$
320 print "Telephone number:";AREA$,TEL$
330 goto 60
```

For more information see FIELD, PUT#, GET#, OPEN and CLOSE.

OPEN OUT # *(Open a file for output)*

OPEN OUT #channel,file$[,attribute]

The OPEN OUT instruction is used to open a sequential file for writing using PRINT#. If this file already exists on the disc it will be erased. *Channel* is a number between 1 and 10 by which the file will be referred to in all subsequent operations. *File$* can be any string holding the name of the new file to be opened. The optional *attribute* allows you to specify the file type to be used. See DIR FIRST$ for more details. Note that any attempt to read a file opened by OPEN OUT will cause an error.

See CLOSE, OPEN IN, POF,LOF,EOF and PRINT#

OPEN IN # *(Open a file for input)*

OPEN IN #channel,file$

OPEN IN is used to open a file for reading. This file is only available for reading, so if you try to write to a file open using OPEN IN, an error will occur. *Channel* denotes a number ranging from 1 to 10 which is used by the instructions INPUT#,LINE INPUT# and INPUT$ (#channel,count) to specify which file is to be read.

See OPEN,CLOSE INPUT# LINE INPUT#,INPUT$(#channel,n), EOF, POF and LOF

OPEN # *(Open a channel to a random file or a device)*

There are four forms of this instruction:

OPEN #Channel,"R",file$ (Opens a random access file)
OPEN #Channel,"MIDI" (Opens a channel to the MIDI interface)
OPEN #Channel,"AUX" (Open a channel to the RS232 port)
OPEN #Channel,"PRT" (Open a channel for the printer) (assumes it's plugged in the parallel port)

Example:

```
10 open #1,"AUX"
20 for I=0 to 10
30 print #1,"STOS BASIC"
40 next X
50 close #1
```

This program prints out ten lines of text on the device connected to the RS232 port. If your printer uses the parallel port change line 10 to:

```
10 open #1,"PRT"
```

Similarly you can input information from a device such as a modem with a line like:

```
30 input #1,A$:print A$
```

When accessing these external devices, all the normal input statements are available for your use, including INPUT$ and LINE INPUT.

See PORT, CLOSE, PUT, GET, FIELD$

CLOSE # *(Close a file)*

CLOSE #channel

This function closes the file associated with a channel. If you forget to close a file after you have finished with it, any changes you have made to the file will be completely ignored.

Example:

```
close #1
```

PRINT # *(Print a list of variables to a file or device)*

PRINT#Channel,variable list

This command is identical to the normal print instruction, but instead of displaying the information to the screen, it outputs it to a file or output device specified by the channel.

Example:

```
print #1,"Hello"
```

As with PRINT you can abbreviate PRINT# to ?#.

Example:

```
? #1,"Hello Again"
```

See also OPEN IN, OPEN OUT, OPEN, PRINT, USING

INPUT # *(Input a list of variables from a file or device)*

INPUT #Channel,variable list

INPUT# reads information from either a sequential file, or a device such as the MIDI interface. The format of the instruction is identical to its screen equivalent. As before it expects each piece of data in the file to be separated by a comma. INPUT can only read up to a maximum of 500 characters worth of data at any one time. If your data is larger than this, you should always use the INPUT$ instruction instead.

LINE INPUT # *(Input a list of variables not separated by a ",")*

LINE INPUT # has two possible formats:

LINE INPUT #Channel,variable list

or

LINE INPUT #Channel,separator$,variable list.

This function is identical to INPUT#, but it allows you to use another character instead of a comma to separate the individual items of data on the disc. If no separator$ character is included, then <Return> is assumed.

INPUT$ *(Inputs a number of characters from a device)*

INPUT$ (#Channel,count)

This reads *count* characters from the device or file connected to channel.

EOF # *(Test for end of file)*

EOF (#Channel)

EOF is a useful STOS Basic function which tests to see the end of a file has been reached at the current reading position. If it has, EOF returns a result of true, otherwise false.

LOF # *(Length of open file)*

LOF(#Channel)

This simply returns the length of an open file. It makes no sense to use this function in conjuction with devices other than the disc.

POF # *(Variable holding current position of file pointer)*

POF(#Channel)

The POF function changes the current reading or writing position of an open file, for example:

pof(#1)=1000

This sets the read/write position to 1,000 characters past the start of the file. Oddly enough POF can be used in this way to provide a crude form of random access when using sequential files! The reason this works is simply that disc drives are inherently random, and all sequential operations are effectively simulated using random access.

FIELD # *(Define record structure)*

FIELD #channel, length1 AS field1$,
length2 AS field2$.....

FIELD allows you to define a record which will be used for a random access file created using the OPEN #channel,"R" command. This record can consist of up to 16 alphanumeric fields and be up to 65535 bytes in length.

Example:

FIELD #1,15 as SURNAME$,15 as NAME$,10 as CODE$,10 as TEL$

See OPEN, GET, PUT, CLOSE

PUT # *(Output record R to a random access file)*

PUT#channel,R

PUT moves a record from the ST's memory into record number R of a random access file. Before use, the contents of the new record should first be placed in the field strings defined by FIELD, using a statement such as:

SURNAME$="HILL"

Although you can write existing records in any order you like, you are not allowed to scatter records on the disc totally at random. This means that if you have just created a file, you can't type in something like:

```
put #1,1
put #1,5
```

In this case, the PUT #1,5 instruction will generate an error, as there are no records in the file with numbers between 1 and 5.

See also OPEN, GET, FIELD$

GET # *(Input record R from a random access file)*

GET #Channel,R

GET reads record number R stored in a random access file opened using OPEN. It then loads this record into the field strings created by FIELD. These strings can now be manipulated in the normal way.

Example:

```
10 open #1,"R","test"
20 field #1,10 as NAME$
30 I=1
40 input "Name?";NAME$
50 if NAME$="" then 90
60 put #1,I
70 inc I
80 goto 40
90 input "Record number?";R
100 if R<0 then close #1 : end
110 get #1,R
120 print NAME$
130 goto 90
```

Note that you can only use GET to retrieve records which are actually on the disc. If you try to grab a record number which does not exist, an error will be generated.

PORT # *(Function to test if channel waiting)*

PORT(#Channel)

The PORT function tests to see if an input device connected to a *channel* is waiting for you to INPUT some information from it.

X=PORT(#channel)

If channel is ready to output some information, then X will be set to -1 (true), and otherwise it will be zero (false).

The printer

There is also a separate set of instructions for use with the printer.

LLIST *(Print part or all of a program on a printer)*

This just lists your program to the printer. The syntax of the LLIST instruction is exactly the same as that of LIST.

Example:

LLIST 10 **Outputs line 10 to the printer.**

LLIST 10-100 **Lists the lines from 10 to 100 to the printer.**

LLISTLists your entire program.

See LIST

LPRINT *(Output a list of variables to the printer)*

As PRINT but sends your data to the printer instead of the screen.

Example:

lprint "Hello"

See PRINT , USING, PRINT#

LDIR *(List a directory to the printer)*

Lists the directory of the current disc to the printer. See DIR, for more details.

LISTBANK *(Print a list of the banks used by your program on the printer)*

Lists the status of all the banks used by the current program using the printer. See LISTBANK

HARDCOPY *(Screen dump)*

This instruction dumps a copy of all the graphics on the screen to the printer. Identical to pressing the Alt+Help keys from the editor. Note that people with Epson compatible printers should first set the correct printer type. Since this requires you to access the ST's inner workings directly, we've included an example routine for this purpose in the technical reference section as an example of the TRAP instruction.

WINDCOPY *(Window dump)*

Unlike HARDCOPY this command prints out the text in the currently open window. As you would expect, it is much faster than the graphics dump produced by HARDCOPY.

Directories

DIR *(Print out the directory of the current disc)*

DIR [PATH$] [/W]

This function lists all the files on the current disc. If the optional path$ is specified, only the files which satisfy a certain set of conditions will be displayed. This path string can contain any one of the following six parts:

- The Name of a drive terminated by a ":"
- The name of a folder to be listed. (Enclosed between two "\" characters)
- A string of characters which will be matched in every filename to be displayed.
- A "*" denoting that any string of up to eight characters will do.
- A "?" which automatically matches with any single character in the filename.
- A "." which separates a filename from an extension.

If the optional /W is added then the files will be listed across the page.

Examples:

```
DIR "A:*.BAS":rem Lists... lists all Basic programs
on the disc.
DIR "\STOS\*.*":rem Lists... lists all files in the folder STOS
DIR "\STOS\*.CR?":rem Lists list all the available
character sets.
```

DIR$ *(Set the current directory)*

This reserved variable can be used to find or change the default directory used for all disc operations, such as loading and saving.

Example:

```
DIR$="\STOS"
DIR (Displays the files in folder STOS)
```

DIR FIRST$ *(Get first file in directory satisifying path name)*

DIR FIRST$(path$,flag)

This function returns a string containing the name and parameters of the first file on the disc which satisfies the conditions in the pathname *path$*. The *flag* contains a number of binary bits which indicate the type of files to be searched for. The format of this flag is:

Bit 0 Normal Read/Write files
Bit 1 Read only files
Bit 2 Hidden files
Bit 3 Hidden system files
Bit 4 Volume labels (The name of the disc)
Bit 5 Folders
Bit 6 Files which have been written to and closed

If you aren't sure which type of files you want to list, you can find all the files on the disc by setting the flag to -1.

If no file exists on the disc matching your specifications, then DIR FIRST$ will return a null string. Otherwise it will hold the following 42 character parameter block.

Characters	Usage
0-12	Filename
13-21	Length of file
22-32	Date file saved
33-41	Time file saved 42 : file type

See DIR NEXT$ for an example of this function in action.

DIR NEXT *(Get the next file satisfying current path)*

DIR NEXT$ returns the next file found using the path specified by DIR FIRST$. It can only be used after a DIR FIRST$ instruction has been executed. The string returned by this function is in exactly the same format as the one generated by DIR FIRST$. As before, if the string returned by the function is empty, then there are no more files in the current path.

Example:

```
new
10 input "Input path$";P$
20 N$=dir first$(P$,-1) : if N$="" then end
30 print "Files matching the path string ";P$
33 print
35 print "Names";space$(8);"Size";space$(5);"Date";
space$(7);"Time";space$(5);"Type"
40 print "=========================================
==========="
50 print N$
60 repeat
70 N$=dir next$
80 print N$
90 until N$=""
```

In order to print a list of the all the files on the disc, simply run this program with a path of "*.*"

Also see DIR FIRST$, PREVIOUS, DIR, DIR$

PREVIOUS *(Sets the current path up one directory)*

This function can be used to move the search path up to the next outer subdirectory.

Example:

```
dir$="\STOS
dir
previous
dir
```

See DIR$

DRIVE *(Variable containing the number of the current drive)*

DRIVE is a variable containing a number representing the drive vou are currently using, with 0 denoting drive A, 1 indicating drive B etc.

Example:

```
print "Current DRIVE is ";drive
drive=1
print "Current DRIVE is ";drive
```

See DRIVE$, DRVMAP

DRIVE$ *(String variable holding current drive)*

This function holds the letter representing the drive.

Example:

```
print "Current drive is ";drive$
drive$="B"
print "Current drive is ";drive$
```

DRVMAP *(Variable holding a list of the drives connected)*

DRVMAP holds a binary number denoting the number of the drives connected. Each binary digit in the number holds the status of one of the drives, starting with bit 0. If the bit at a particular position is set to one, then the appropriate drive is attached to the computer. So:

```
Bit 0 = Drive A
Bit 1 = Drive B
Bit 2 = Drive C
```

Example:

```
print bin$(drvmap,26)
```

Note that, drvmap always assumes a minimum of two drives, even if you're only using a standard ST.

DFREE *(Variable containing the free space on the current disc)*

DFREE holds the amount of free space remaining on a disc.

```
print dfree
```

MKDIR *(Create a folder)*

MKDIR folder$

This function creates a folder with the name *folder$*.

Example:

```
mkdir "TEST
dir
```

RMDIR *(Delete a folder)*

RMDIR folder$

RMDIR deletes an empty folder from the disc.

Example:

```
rmdir "TEST
dir
```

KILL *(Erase a file from the disc)*

KILL file$

This function deletes a file with the name *file$* from the current disc. If *file$* contains the characters "*" or "?" a series of files will be erased. You should be very careful when you use this function as anything you kill is wiped from the disc permanently.

RENAME *(Rename a file)*

RENAME old$ TO new$

The RENAME function allows you to change the name of a file. *old$* refers to the existing name, and *new$* to the new name. If a file already exists with the new name you have chosen, an error will be generated.

Example:

```
rename "DUMP.ACB" to "EXAMINE.ACB"

This renames the DUMP.ACB accessory.
```

Trigonometric functions

DEG *(Convert an angle expressed in radians to degrees)*

DEG converts angles expressed in radians into the form of degrees. A degree is approximately equal to 57 radians.

Example:

```
print DEG(90)
5156.62015618
```

See RAD

RAD *(Convert a radian expressed in degrees to radians)*

RAD converts angles expressed in degrees into radians. A radian is approximately equal to 57 degrees.

Example:

```
print RAD(5156.62015618)
90
```

See DEG

These functions all use so called radian measure. One radian is equal to 360/2*PI or approximately 57 degrees.

SIN *(Sine)*

SIN(angle)

Calculates the sine of the *angle*. Note that this function always returns a floating point number, so if you wish to assign the return value to a variable, this must always be of the type double precision.

Examples:

```
P#=sin(pi/2)
print sin(pi/4)
```

See ASIN,HSIN and PI

COS *(Cosine)*

COS(angle)

Returns the Cosine of the number in *angle* as a floating point number. All angles are measured in radians.

```
Q#=cos(pi/2)
print cos(pi/4)
```

See ACOS, HCOS and PI

TAN *(Tangent)*

TAN(angle)

Generates the Tangent of the *angle*.

Examples:

```
R#=tan(pi/3)
print tan(pi/4)
```

See ATAN, HTAN and PI.

ASIN *(Arc sine)*

ASIN(number)

This function takes a number between -1 and +1 and calculates the angle in radians which would be needed to generate this value with SIN.

So if X#=SIN(ANGLE) then ANGLE=ASIN(X#).

Examples:

```
A#=asin(1)
print asin(0.5)
```

See SIN, HSIN(), PI()

ACOS *(Arc cosine)*

ACOS(number)

ACOS reverses the action of COS in the same way that ASIN inverts the SIN function.

Example:

```
B#=acos(1)
print acos(0.5)
```

See COS, HCOS(), PI()

ATAN *(Arc tangent)*

ATAN(number)

Generates the arctan of *number*. See TAN ,HTAN, PI

Example:

```
C#=atan(0.5)
print atan(0)
```

HSIN *(Hyperbolic sine)*

HSIN(angle)

Returns a double precision number denoting the hyperbolic sine of an *angle*.

See SIN, ASIN

HCOS *(Hyperbolic cosine)*

HCOS(angle)

Returns a double precision number denoting the hyperbolic cosine of *angle*.

See also COS, ACOS

HTAN *(Hyperbolic tangent)*

HTAN(angle)

Returns a double precision number denoting the hyperbolic tangent of *angle*.

See also TAN, ATAN

PI *(A constant π)*

This function returns the number called PI which represents the result of the division of the diameter of a circle by the circumference. PI is used by most of the trigonometric functions to calculate angles.

Mathematical functions

LOG *(Logarithm)*

LOG(y#)

This function returns the logarithm in base 10 (log10) of Y# as a double precision number.

Examples:

```
print log(10)
V#=log(100)
```

LN *(Natural Logarithm)*

LN(Y#)

LN calculates the natural or naperian logarithm of Y#.

Examples:

```
print ln(10)
R#=ln(100)
```

The action of LN is exactly opposite to that of EXP

EXP *(Exponential function)*

EXP(Y#)

Returns the exponential of Y# as a double precision number.

Examples:

```
print exp(1)
TEST#=exp(ln(100))
```

=SQR *(Square root)*

X=SQR(Y)

SQR calculates the number which must be multiplied by itself to get the value of *Y*.

X=sqr(4)

Returns a value of 2 in X.

Example:

```
10 input "input a positive number ";N
20 print "The square root of ";N;" is ";sqr(N)
30 goto 10
```

ABS *(Absolute value)*

ABS(y)

ABS returns the absolute value of *y*, taking no account of the sign of the number.

Example:

```
print abs(-1),abs(1)
1 1
```

INT *(Convert floating point number to an integer)*

INT(y#)

This rounds down the decimal value of *y* and converts it into a whole number.

Examples:

```
print int(1.25)
1
print int(-1.25)
- 2
```

SGN *(Find the sign of a number)*

SGN(y)

This allows you to find the sign of the number or expression in *y*. The function returns one of three possible values:

```
-1 if Y is negative
 0 if Y is zero
 1 if Y is positive

10 input X
20 if sgn(X)=-1 then print "Number is negative"
30 if sgn(X)=0 then print "Number is zero"
40 if sgn(X)=1 then print "Number is positive"
50 goto 10
```

MAX *(Get the maximum of two values)*

MAX(x,y)

The MAX function compares two expressions and returns the largest. These expressions can be composed of numbers or strings of characters, providing you don't try to mix different types of expressions in one instruction.

So

```
print max(10,4)
is ok returning10
```

and

```
print max("Hello","Hi")
is also legal returning Hi
```

But you can't however use something like:

```
print max(10,"HI")
```

See MIN

MIN *(Return the minimum of two values)*

MIN(X,Y)

MIN returns the smallest of the two expressions you specified. These expressions

can consist of strings, integers or real numbers. However you must only compare values of the same type.

Examples:

```
print min(10,4)
4
print min("Hello","Hi")
Hello
```

See MAX

SWAP *(Swap the contents of two variables)*

SWAP(X,Y)

This swaps the data between any two variables of the same type. For instance:

```
new
10 A=1 : B=100
20 C$="Left" : D$="Right"
30 print A,B,C$,D$
40 swap A,B
50 swap C$,D$
60 print A,B,C$,D$
```

DEF FN *(Create a user-defined function)*

DEF FN is a useful function which enables you to create your own user-defined functions for use within a STOS Basic program.

The syntax of this function is:

DEF FN name [(variable list)]=expression

name is the name of the function you wish to define.
variable list can be any list of variables separated by commas. These variables are local to the function. Any variables you use in the function will be automatically substituted for the appropriate local variables whenever necessary. Also note that variables of different types can be mixed within a single function.

FN *(Call a user defined function)*

FN name [(variable list)]

FN is used to execute a function defined by DEF FN.

Examples:

```
new
10 def fn SQ (X)=X*X
20 input "Input a number";I
30 print "The square of ";I;" is ";fn SQ (I)
40 goto 20
```

```
new
10 def fn DEG (R)=R*pi/180
```

```
20 print sin(fn DEG (45))

new
10 def fn SEGMENT (A$,X,Y)=mid$(A$,X,Y)
20 print fn SEGMENT ("Hello",2,3)
```

See how we've always placed the DEF FN statement in the program before it is used.

RND *(Random number generator)*

RND(y)

RND is used to generate a random integer between 0 and *y* inclusive. If *y* is less than zero, RND will return the last value it produced. This is very useful when debugging a program.

Examples:

```
10 plot rnd(640/divx-1),rnd(400/divy-1) 20 goto 10

print "Dice throw is a ";rnd(6)
```

LET *(Load some information into a variable)*

Used to assign a variable to a specific value. The use of LET is always optional and can be omitted whenever you like.

Examples:

```
let A=1
let A$="Hello"+" "+"there"
```

FIX *(Set precision)*

FIX(n)

This procedure fixes the precision of any real numbers which are to be printed on the screen. There are three possibilities.

If $0<n<16$ then *n* denotes the number of figures to be output after the decimal point.

If $n>16$ the printout will be proportional and any trailing zeros will be removed.

If $n<0$ then all floating point numbers will be displayed in exponential format, and the absolute value of *n* (ABS(*n*)) will determine the number of digits after the decimal point.

Examples:

fix (2):print PI	**Limits the number to two digits after the point.**
fix(-4):print PI	**Forces exponential mode with four figures after the point.**
fix(16):print PI	**Reverts to the normal mode.**

String Functions

UPPER$ *(Convert to upper case)*

UPPER$(n$)

This function converts the string in n$ into upper case (capitals).

Example:

```
print upper$("StoS BaSic")
STOS BASIC
```

Do not confuse this with the editor command UPPER.

LOWER$ *(Convert to lower case)*

LOWER$(n$)

LOWER$ translates all the characters in *n$* into lower case.

```
print lower$("Stos Basic")
stos basic
```

This function should not be confused with the editor directive LOWER.

FLIP$ *(Invert String)*

FLIP$(n$)

FLIP$ reverses the order of the characters in the string *n$*.

Example:

```
print flip$("STOS Basic")
cisaB SOTS
```

SPACE$ *(Create a string full of spaces)*

SPACE$(n)

SPACE$ generates a string containing n spaces.

Example:

```
print space$(20)"      : Spaces"
                    : Spaces
```

STRING$ *(Create a string full of a$)*

STRING$(a$,n)

STRING$ creates a string of *N* characters using the first character of the string *a$*.

Example:

```
print STRING$("The cat sat on the mat",10)
TTTTTTTTTT
```

Note that STRING$(" ",X) is identical to SPACE$(X)

CHR$ *(Return Ascii character)*

CHR$(n)

Creates a string containing the character with the Ascii code *N*.

Example:

```
print chr$(66)
B
```

ASC *(Get Ascii code)*

ASC(a$)

This returns the Ascii code of the first character of the string in *a$*.

Example:

```
print asc("B")
66
```

LEN *(Get length of string)*

LEN(a$)

LEN calculates the current length of a string of characters held in *a$*. All the characters of a string are counted, even if they are not visible on the screen. So LEN(CHR$(27)+CHR$(27)) will give the number 2.

Example:

```
print len("12345678")
8
```

Do not confuse with LENGTH.

VAL *(Convert a string to a number)*

VAL(x$)

VAL returns the value of a number stored in the string *x$*. If *x$* does not contain a number then VAL will be zero.

Example:

```
10 input "Input a number";A$
20 A#=val(A$)
30 if A#=0 then print A$;" is NOT a number" : goto 10
40 print "The square root of ";A#;" is ";sqr(A#)
```

STR$ *(Convert number to string)*

STR$(n)

This function converts a number in a string of characters. STR$ can be very useful since some functions, such as CENTRE, do not allow you to use numbers as an parameter.

Example:

```
centre "Memory left is "+str$(free)+" bytes"
Do not confuse STR$ with STRING$
```

TIME$ *(Get time)*

TIME$ holds a string containing the current time in hours, minutes and seconds using the format "HH:MM:SS"

```
10 time$="10:50:00"
20 print time$
30 goto 20
```

This string is updated by STOS once every 50th of a second. See also TIMER, DATE$

DATE$ *(Get Date)*

This stores the current date as a string of characters in the format "DD/MM/YYYY" where DD represents the day, MM the month and YYYY the year.

Example:

```
print date$
```

Note that if you don't have a clock card fitted, this date must be set directly using a statement like:

```
DATE$="09/06/1988"
```

See also TIMER and TIME$

FILESELECT$ *(Select a file)*

This is a very powerful feature which enables you to call up a fancy dialogue box to select one the files on the disc.

The syntax of this function is:

f$=FILE SELECT$(path$ [,title$ [,border]])

path$ can be any string containing the search pattern which will be used to display the possible files.

title$ is a string containing the title of the dialogue box.

border is a number from 1 to 16 denoting the border style which is to be used.

After completion of the dialogue, FILE SELECT$ returns either the name of the file or an empty string if the QUIT option was chosen.

Examples:

```
new
10 X$=file select$("*.*")
20 print X$

print file select$("*.BAS")
```

See also FSAVE and FLOAD.

Machine level instructions

HEX$ *(Convert number to hexadecimal)*

HEX$(n)

HEX$ converts a number into a string of characters in hexdecimal notation. There are two possible formats of this instruction.

X$=HEX$(x)
Loads *x$* with number *x* expressed in base 16

X$=HEX$(x,n)
Loads *x$* with the first *n* digits of *x*, where *n* can range from 1 to 8.

Examples:

```
print hex$(colour(0))
print hex$(65536)
$10000
print hex$(65536,8)
$00010000
```

BIN$ *(Convert number to binary string)*

BIN$(x)

BIN$ generates the string of binary digits equivalent to the number x. As with HEX$, you can choose whether to generate all the digits or only a few.

Example:

```
print bin$(255)
%11111111
print bin$(255,16)
%0000000011111111
```

The precise syntax of the BIN$ function is:

x$=BIN$(x) Where *x* is the number to be converted to binary.

or

x$=BIN$(x,y) When *x* is the number to be used, and *y* the number of digits in the string which will be loaded into *x$*. *y* can range between 1 and 31

ROL X,Y *(Rotate left)*

ROL is a Basic version of the ROL instruction from 68000 assembly language. The effect to to take the binary representation of a number in *y*, and rotate it left by *x* places.

Example:

The number 136 is represented in binary by:

%10001000

Type in:

```
X=136
rol.b 1,X
```

This will give the number 17 or binary %00010001

As you can see, the entire number has been shifted to the left, with the highest 1 being rotated into the lowest position. The reason for the ".b", is to instruct STOS to treat this number as an 8-bit byte. You can also specify the sizes ".W" (word) and ".L" (long word).

Note that this procedure expects the number to be shifted to be held in a simple variable and not an expression.

Examples:

```
A=1
rol 1,A
print A
2

A=32768
rol.w 2,A
print A
1
```

If ROL is used without ".B",".W",or".L" then ".L" is assumed. Providing you use reasonably sized numbers ROL can be effectively considered as a very fast way of multiplying a number by a power of 2.

ROR *(Rotate right)*

ROR X,Y

This is similar to ROL but rotates the number in the opposite direction.

Example:

```
A=8
ror 1,A
print A
4
```

Note that ROR can be used as a very fast way of dividing a number by a power of two.

BTST *(Test a bit)*

BTST(X,Y)

This function allows you to test the binary digit at position x in the variable Y. As with the functions ROR and ROL, y must be a single variable and not an expression. If the bit at *x* is set to 1, then the value of BTST will be true, otherwise it will be false.

Example:

```
new
10 input "Enter a number";N
20 input "Enter a bit to be tested";B
30 if B<0 or B>31 then end
40 print "Bit Number ";B
50 if btst(B,N) then print " is a one " else print "is a zero"
60 print bin$(N,32)
80 goto 10
```

See also BCHG, BCLR, BSET

BSET *(Set a bit to 1)*

BSET(x,y)

BSET sets the bit at position *y* to 1 in the variable *x*. As before *x* must be a simple variable rather than an expression.

Example:

```
A=0
bset 8,A
print A
256
```

BCHG(x,y) *(Change a bit)*

BCHG(x,y)

This procedure changes bit number *y* in the variable *x*. If this bit is currently a 1 then the new value will be a zero, and vice versa.

Example:

```
A=0
bchg 1,A
print A
2

bchg 1,A
print A
0
```

BCLR *(Clear a bit)*

BCLR(x,y)

BCLR sets bit number *y* in variable *x* to a zero.

Example:

```
A=128
bclr 7,A
print A
0
```

PEEK *(Get byte at address)*

PEEK(address)

This function returns the 8 bit byte stored at *address*. Technically-minded readers will be interested to note that PEEK gets information from the ST's memory while in supervisor mode. This means that you can happily type in something like:

```
print peek(0)
```

POKE *(Change byte at address)*

POKE address,x

Loads *address* with the number from 0-255 stored in *x*. You may use this function to change the contents of any part of the ST's memory. But be warned that this function is dangerous. If you poke around indiscrimantly you will almost certainly crash the ST completely.

Example:

```
poke physic+1000,255
Pokes a blob on the ST's screen
```

DEEK *(Get word at address)*

DEEK(address)

This function reads the two-byte word at *address*. This address MUST be even or an address error will occur.

As with PEEK, you can use DEEK to access any part of the ST's memory including the sections that are normally inaccessable.

Example:

```
print deek(0)
```

DOKE *(Change word at address)*

DOKE address,value

DOKE loads a two byte number between 0 and 65535 into *address*. In knowledgeable hands this function can be very useful, but since even the best of us make mistakes, you should always remember to save a copy of your programs to the disc before attempting to use this function in a new routine.

Example:

```
doke physic+1000,65535
```

LEEK *(Get long word at address)*

LEEK(address)

The LEEK function returns the four-byte long word stored at *address*. Like DEEK, the address used with this function must always be even. Note that if bit 31 of the contents of *address* is set, the number returned by LEEK will be negative.

Example:

```
print leek(0)
```

LOKE *(Change long word at address)*

LOKE address,number

LOKE loads *address* with a four-byte long word specified by *number*.

Example:

```
loke physic+10000,$FFFFFFFF
```

Indiscriminate use of this function can lead to the ST crashing completely, so take care.

VARPTR *(Get address of a variable)*

VARPTR(variable)

This function returns the location in the ST's memory of a variable. Each of the different types of variables are stored in a different way.

Integers: VARPTR returns the location of the value of the variable.

Example:

```
A=0
loke varptr(A),1000
print A
1000
```

Real numbers: VARPTR returns the location of two long words which contain the value of the variable in the IEEE double precision format.
Strings: VARPTR points to the first character of the string. Since STOS Basic does not end its strings with a character 0, you must obtain the length of the string using something like: DEEK(VARPTR(A$)-2), where *A$* is the name of your variable. You could also use LEN(A$) of course.

COPY *(Copy a memory block)*

COPY start,finish TO destination

This command is used to rapidly move large sections of the ST's memory from one

place to another. *Start* is the address of the start of the block of memory to be moved, and *finish* is the address of the end. *Destination* points to the first memory location of the destination.

Note that all these addresses MUST be even.

Example:

copy logic,logic+10000 to logic+10000

This copies one part of the screen to another.

FILL *(Fill memory block with a longword)*

FILL start TO finish,longword

FILL copies a specific long word into a section of memory.

start is the beginning of the block and *finish* the end. *longword* is the data which will be copied into each set of four memory locations between *start* and *finish*. Note that it's also possible to use the number of memory BANK as the start or finish location.

Example:

fill logic to logic+32000,$22334455 Displays a series of lines on the screen.

fill 1 to 2,0 Fills bank 1 with 0.

Incidentally, if start and finish are specified as an address, these values MUST be even.

=HUNT *(Find a string in memory)*

X=HUNT(start TO end, A$)

This command is used to allow you to search through the ST's memory for a specific character string.

start is the position in the ST's memory of the start of the search, and end is the address of the end. On completion of this routine X will hold either 0 (if the string in *A$* was not found) or the location of *A$*.

WAIT *(Wait in 50ths of a second)*

WAIT x

This function suspends a STOS Basic program for *x* 50ths of a second. Any functions which use interrupts, such as MOVE and MUSIC will continue to work during this period, with the sole exception of ON MENU GOTO.

Example:

wait 50

This waits for one second.

TIMER *(Count in 50ths of a second)*

TIMER is a reserved variable which is incremented by one every 50th of a second. Here is a small example showing how this is used.

Example:

```
new
10 print "Started"
20 timer=0
30 if timer<500 then goto 30
40 print "Finished"
```

NOT *(Logical NOT operation)*

NOT(x)

This function changes every binary digit in a number from a 1 to a 0 and vice versa. Since True =-1 and False=0, NOT(True)=False.

Examples:

```
print not(-1)

new
10 if not(true)=false then print "False"
```

Miscellaneous instructions

REM *(Remark)*

Any text typed in after a REM statement will be completely ignored by STOS Basic. You can therefore use this instruction to place comments at appropriate points in your programs. Note the apostrophy character; ', is an abbreviation for rem.

Example:

```
10 rem This program does absolutely nothing
```

DATA *(Place a list of data items in a STOS Basic program)*

The DATA statement allows you to incorporate lists of useful data directly inside a Basic program. This data can be loaded into a variable using the READ instruction. The format of the DATA statement is:

DATA variable list.

Each variable in the list is separated by a comma.

Example:

```
10 data 1,2,3,"Hello"
```

Unlike many other Basics, the STOS version of this instruction also allows you to use expressions involving variables. So the following lines of code are perfectly acceptable

```
10000 data $FF50,$890
10010 data %1111111111111,%1101010101
10020 data A
10030 data A+3/2.0-sin(B)
10040 data "Hello"+"There"
```

Note that the A in line 10020 will be input as the contents of variable A, and not the Ascii character A. Similarly the expression at line 10030 will be evaluated during the READ operation using the current values of A and B.

Incidently, DATA must always be the only instruction on a line.

See READ, RESTORE.

READ *(READ some data from a DATA statement into a variable)*

READ list of variables

READ allows you to input some data stored in a DATA statement into a list of variables. It starts off with the first data statement in the program, and then reads each subsequent item of data in turn. As you might expect, the variable used in each READ instruction must always be of the same type as the information stored in the current DATA statement.

Example:

```
new
10 for I=1 to 10
20 read A
30 next I
40 data 1
50 data 2,3
60 rem
70 data 4,5,6,7,8
80 data 9,10
```

Note that STOS Basic also lets you use complex expressions in a DATA statement.

Example:

```
new
10 T=10
20 read A$,B,C,D$
30 print A$,B,C,D$
40 data "String",2,T*20+rnd(100),"STOS"+"Basic"
```

READ uses a special pointer to determine the location of the next piece of data to be input. This pointer can be changed at any time in the program using the RESTORE instruction.

See RESTORE, DATA.

RESTORE *(Set the current READ pointer)*

RESTORE line

This instruction changes the line number at which a subsequent READ operation

will expect to find the next DATA statement. There are two forms of this instruction.

RESTORE line	Set start of DATA statements from *line*
RESTORE expression	Calculate line number and set read pointer to this line.

If a data statement does not exist at the line specified by RESTORE, an appropriate error message will generated.

Example:

```
new
10 restore 1000+language*10
20 read A$
30 print A$
40 end
1000 data "English"
1010 data "Francais"

francais
run
Francais
english
run
English
```

See also READ, DATA

TRUE *(Logical TRUE)*

This function returns a value of -1, which is used by all the conditional operations such as IF...THEN and REPEAT...UNTIL to denote true.

```
10 if -1 then print "Minus 1 is TRUE"
20 if TRUE then print "and TRUE is ";TRUE
```

See FALSE, NOT

FALSE *(Logical FALSE)*

Whenever a test is made such as X>10, a value is produced. If the condition is true then this number is -1, otherwise it is zero. The FALSE function therefore corresponds to a value of 0.

```
Print FALSE
0
```

See TRUE.

11 Writing a game

There are no real rules on how you should go about programming a game, but there are many points which can help in its design and development.

Planning

The most important part of game writing is the initial specification and its planning. First decide what you want the game to do then layout every detail so that you have a complete picture of your desired end product. If you don't plan the game it will take much longer to write than if you had. Remember: Fools rush in where angels fear to tread.

Planning techniques

The intial idea may come fairly quickly – but the more interesting features may take a while to come. Use a thesaurus to help you find more references to your game idea. We used one while trying to think up a name for Orbit. Starting with the word *ball* we soon found an apt and original name.

Say you wanted to create a game to be called Haunted House. You could start by looking up ghost or ghoul, and then move from section to section gathering together useful ideas which you may be able to incorparate into your games.

Once the ideas for the game have been laid out on paper, you can then start modularising sections. This means looking at your game idea and deciding which parts are independent areas that don't rely on other sections of the game to work. Take for example the game Orbit: The ball that bounces around the screen would be one module, the player's bat another and the bricks a further one.

Another aspect of planning are the screen designs. Screens in the game must be accurate and designed to use STOS Basics commands to their best benefit. A badly laid-out screen will cause numerous problems during programming and a screen re-vamp will probably be necessary wasting valuable time.

Programming

This section of the game development will take most of the time and is a very critical stage. Programming is an art, requiring patience and logical thinking. You will find that your skill will improve as you write more and more programs. The emphasis with game programming is speed – a super animated space game is no use if the response to the player is too slow.

The key word in programming is structure. All structured programs should be:

- Readable — Easy to follow logic
- Reliable — They do what was intended
- Adaptable — For possible later modifications

Write the modules from the planning section as subroutines, thus creating a

Adding graphics

Computer graphics can transform simple game ideas into professional, well-presented products. The graphics help to create a new world of reality and thus complement the programmer's skill. The major problem with adding graphics to a game is usually the fact that the programmer cannot draw very well. This has therefore produced a new wave of jobs in the games industry for graphic artists. Get help from a friend who is good at art if your own talents don't stretch very far.

Graphics can be split into sections:

Pictures

STOS Basic can load in files saved from Neochrome and Degas. Both these programs are widely used and are exceptionally well-designed.

Geometry

This is more a mathematical form of graphics and you really don't need any artistic qualities. Using STOS Basic's drawing commands you can create images on a coordinate based system.

Sprites

These are very important in the production of a game and can give great animation effects that will bring your game to life. The size and number of sprites are important factors to consider when writing a game.

Techniques

You will find that there are various ways to program a single situation. In this section we will list various techniques that explain how to get the very best performance from STOS.

Speedy sprites

Most games require a lot of speed so that numerous sprites can be whizzed around the screen. The sprites in STOS Basic are software sprites – which means that the computer has to do all the work of calculating where on the screen they must go and also posistion them. The main thing to remember is that small sprites can be moved around faster than large ones.

So when you're deciding what size sprites to have in your game, ponder on the following points:

Numbers — If you only have a couple they can be large. But if you intend on using all 15 they will have to be small. If you need many sprites in a game then use the copy techniques discussed in Chapter 4.

Size — As we said above, the bigger the sprites are the slower they move. If a game has missiles in it these would be small narrow sprites which take up little of the computer's time.

Adding graphics

Computer graphics can transform simple game ideas into professional, well-presented products. The graphics help to create a new world of reality and thus complement the programmer's skill. The major problem with adding graphics to a game is usually the fact that the programmer cannot draw very well. This has therefore produced a new wave of jobs in the games industry for graphic artists. Get help from a friend who is good at art if your own talents don't stretch very far.

Graphics can be split into sections:

Pictures

STOS Basic can load in files saved from Neochrome and Degas. Both these programs are widely used and are exceptionally well-designed.

Geometry

This is more a mathematical form of graphics and you really don't need any artistic qualities. Using STOS Basic's drawing commands you can create images on a coordinate based system.

Sprites

These are very important in the production of a game and can give great animation effects that will bring your game to life. The size and number of sprites are important factors to consider when writing a game.

Techniques

You will find that there are various ways to program a single situation. In this section we will list various techniques that explain how to get the very best performance from STOS.

Speedy sprites

Most games require a lot of speed so that numerous sprites can be whizzed around the screen. The sprites in STOS Basic are software sprites – which means that the computer has to do all the work of calculating where on the screen they must go and also posistion them. The main thing to remember is that small sprites can be moved around faster than large ones.

So when you're deciding what size sprites to have in your game, ponder on the following points:

Numbers — If you only have a couple they can be large. But if you intend on using all 15 they will have to be small. If you need many sprites in a game then use the copy techniques discussed in Chapter 4.

Size — As we said above, the bigger the sprites are the slower they move. If a game has missiles in it these would be small narrow sprites which take up little of the computer's time.

Scrolling the screen

When using the SCROLL command you must be aware of the limitations caused by horizontal scrolling. Because of the vast number of calculations that the computer has to make while scrolling the screen horizontally, it leaves little time for anything else. The fastest way to scroll the screen left or right is to scroll it on 16 bit (word) boundaries by steps of 16 pixels.

Another point to emphasise is that the larger the area to scroll, the slower the scroll speed.

Collisions

When a game is running in full swing it is imperative that your program is checking collisions as often as possible. If you check only once a second in a shoot-'em-up style game then missiles will fly past aliens without killing them. Using the SET ZONE command you can set up various areas of the screen and then ask the computer which zones your sprites are in. This saves a lot of work and is a very powerful feature.

Examining code

If you feel that you cannot understand the best way to link together commands, it's a good idea to follow through the games listings supplied with STOS. All three games were written by the author of STOS Basic so they are prime examples of well written code. Use the SEARCH command to find examples of commands. By reading and examining this code you will learn various short cuts and techniques.

Optimising your programs

When your program is near to completion you may wish to save memory and increase speed. Here are a couple of examples to show you how to optimise your code.

```
10 for A=1 to 10
20 print A
30 next A
```

This can be optimised to:

```
10 for A=1 to 10:print A:next A
```

The new line will save memory because lines 20-30 are not required and the loop speeds up. The commands are all related, being enclosed as a loop, so it makes sense to group them on to a single line.

The line:

```
10 A=A+1
```

can be optimised to:

```
10 inc A
```

Here we see the use of the INC command rather than the standard Basic A=A+1 expression. It saves memory and increases speed.

Appendix A

Error messages

An error occurs when STOS Basic cannot continue with the program and thus reports this fact to you with a brief statement describing what is causing the problem. Errors can also be generated when commands are typed in direct mode.

Many of the errors are obvious and the statement does its job informing you, but some are slightly more cryptic and need a little more explanation – hence the need for an error appendix.

The errors are listed in alphabetical order so that you can find your entry easily and each errors corresponding code is listed with it. This code is created and stored in ERRN.

Error name	Error code

Address error 32
An odd memory address or invalid address has been accessed using the peek and poke commands.

Animation declaration error 58
The ANIM string command has not been properly set.

Array already dimensioned 28
An array has been re-dimensioned at the error line.

Bad date 55
The user has tried to set the date with illegal values using the DATE$ function.

Bad file format 1
A file to be loaded cannot be recognised by STOS as it is not of the correct format.

Bad filename 53
A filename has been used in an input/output procedure which is not legal. An example of this would be LOAD"".

Bad screen address 43
A screen address has been used which is invalid for a proper screen start address. The address must be on a 256 byte boundary.

Bad time 54
The user has attempted to set an illegal time using the TIME$ function.

Bank 15 already reserved 80
This bank is already reserved and must be erased if you wish to reserve it for another purpose.

Bank 15 is reserved for menus 81
Menus are used in the current program and thus you cannot use this bank for anything else.

Break 17
You have pressed Control +C. If you were in a program then STOS returns you to the editor mode.

Bus error 31
An internal error has occurred possibly due to incorrect addressing using the peek and poke commands.

Can't continue 7
STOS cannot continue from the previous break. This mainly happens when a program is stopped and a line is altered thus resetting all variables.

Can't renum 11
STOS has attempted to renumber a section of your program and this action would result in a conflict of line numbers.

Character set not defined 73
A character set has been referenced which does not exist.

Character set not found 78
You have tried to access a character set which does not yet exist.

Direct command used 15
A command which is only available from direct mode has been used within the program.

Disc error 52
The Atari ST returns TOS disc errors back to STOS and when it's not too sure exactly what error has occurred it will produce this error.

It's best to make sure your drive is connected, the disc is valid and the command you processed was legal.

Disc full 51
The disc has run out of space.

Disc is write protected 50
STOS cannot write out information to the current disc because it is physically write protected. Move the tab on the disc, or use another disc.

Division by zero 46
A number has been divided by zero and cannot be handled by STOS Basic.

Drive not connected 83
The current drive is not available. Check your leads and power.

Drive not ready 49
A disc drive is not ready for use.

End of file 64
The end of a file on a disc has been reached.

Extension not present 84
This occurs when you try to run a program which incorporates a new STOS Basic command without loading the relevant extension file first.

Field too long 66
The size of the record you have created with FIELD is greater than 65535 bytes. It's also possible that you have used more than the maximum of 16 fields.

File already closed 63
An attempt to close a file is aborted because it is already closed.

File already open 62
An attempt to open a file is aborted because it is already open.

File not found 48
You have tried to load or open a file for reading and it is not on the current disc.

File not open 59
The program is trying to transfer data to or from a file but the file has not been opened.

File type mismatch 60
A file command has been used which does not correspond with the correct filing system. The error would occur when you try and use the GET and PUT statements on a sequential file.

Flash declaration error 67
The FLASH command has been called incorrectly.

Follow too long 9
STOS has been told to trace too many parameters.

For without next 22
A FOR command does not have its mandatory NEXT instruction listed later in the program.

Illegal direct mode 14
A command input in direct mode is not recognised by STOS.

Illegal function call 13
You have tried to use a function with an illegal set of parameters.

Illegal instruction 82
When STOS is running a machine-code program this error will occur if it finds that the code is invalid.

Illegal negative operand 47
Some functions cannot process negative numbers, for example SQR(-1).

Illegal user-function call 40
The list of parameters you input does not match the list you specified in the DEF FN command.

In/out error 16
An error has occurred during an input/output operation.

Input string too long 61 + 65
An incoming string is too long for a dimensioned variable. Or you may have tried to INPUT # a line more than 500 characters long.

Line too long 6
You have attempted to enter a line more than 700 characters long. STOS can cope with many things but a line this size is rather excessive and poor programming style.

Memory bank already reserved 41
An attempt to reserve a memory bank has failed because it has already been reserved.

Memory bank not defined as screen 42
A command has accessed a memory bank which must be reserved as screen and thus cannot find the information required.

Memory bank not reserved 44
A memory bank has been accessed and is not reserved for any use.

Menu not defined 79
The MENU ON command has been called but no menu has yet been set up.

Movement declaration error 57
The MOVE instruction has not been set correctly.

Music not defined 75
Music cannot be played because there isn't a tune in memory.

Next without for 23
STOS has come across a NEXT instruction which has no FOR. Thus STOS does not know where to loop back to.

No data on this line 33
The RESTORE instruction has tried to restore a line of data. In this case the line did not include a data command.

No more data 34
The READ statement cannot get any more data because all of the DATA lines have been read. In other words, you're out of data.

No more text buffer space 74
If you open over 10 windows the size of a full screen in either mode 1 or mode 2 then the space reserved for the data in each window gets used up and causes this error.

Non declared array 18
An array has been referenced which has not been set up with the DIM instruction.

Not done 0
A procedure has been attempted but due to some condition the job was not carried out. Quitting the file selector and returning to the editor is an example of this error.

Out of memory 2 + 8
STOS has no more memory left for allocation. Take out all accessories and excess programs to free more memory.

Overflow error 21
A calcultion has exceeded the size of a variable.

Pop without gosub 37
The POP instruction cannot be executed outside of a subroutine.

Printer not ready 10
The printer is not on line so STOS cannot output any data. Check all connections and the power switch of the printer.

Repeat without until 26
A REPEAT instruction exists but has no corresponding UNTIL.

Resolution not allowed 45
This occurs on high-resolution monitors when the MODE instruction is used. It happens on colour monitors when you try to enter high resolution.

Resume without error 38
A RESUME instruction cannot be executed unless an error has occurred.

Return without gosub 36
The program has reached a RETURN instruction but no GOSUB has been used.

Scrolling not defined 86
The SCROLL command has been used but STOS does not have the information necessary to scroll the screen. See DEF SCROLL.

Search failed 5
A string has been searched for in the current program but STOS found no reference to it.

Sprite error 56
Parameters for a SPRITE command have been set which do not fall inside the required limits.

String is not a screen block 87
A string has been used in the SCREEN$ command which has not been designed as a sprite block string.

String too long 30
A string has exceeded the limit of 65000 characters.

Subscript out of range 85
A subscript has been accessed which is not dimensioned to the called size. Here is an example: DIM A$(10):A$(12)="HELLO"

Syntax error 12
The syntax (grammar) of the error line or statement is not correct. You must look up the correct syntax in the manual or in the reference card.

System character set called 77
You have attempted to replace a system character set with a custom character set.

System window called 76
The system windows have been used in one of the window commands. These windows are 0, 14 and 15.

This line already exists 4
The Auto function reports this error when it comes across a line which is already in your program.

This line does not exist 3
This error occurs when you have tried to delete a line which does not exist so the delete operation is aborted.

Too many gosubs 35
STOS cannot store any more RETURN addresses.

Type mismatch 19
An illegal value has been assigned to a variable. For example: A$=12 should read A$="12".

Undefined line number 29
This error will happen when you try to GOTO, GOSUB or RESTORE a line which does not exist in the program.

Until without repeat 27
The UNTIL instruction has no repeat command listed later in the program.

User function not defined 39
A user function has been accessed which has not been set up using DEF FN.

Wend without while 25
A WEND instruction has been encountered without a matching WHILE command.

While without wend 24
The WHILE instruction has no mandatory WEND instruction listed later in the program.

Window already opened 69
An attempt to open a window has failed because it is already open.

Window not opened 70
You have referenced a window which does not exist.

Window parameter out of range 68
One of the window's parameters is not valid and must be set to a legal value.

Window too large 72
A window cannot be opened because it is too big.

Window too small 71
An attempt to open a window has failed because it is too small. The minimum size is 3x3.

Appendix B

Creating a runtime disc

The follow-procedure will allow you to create a disc from which you can boot any STOS Basic program without having to load STOS Basic first.

1 The first thing to do is format a blank disc and then load up STOS Basic.

2 Load in the accessory STOSCOPY.ACB with the command

```
accload "STOSCOPY.ACB"
```

Press the HELP key and select the STOSCOPY accessory by pressing the appropriate function key. This accessory will now copy the required files from your STOS Basic master disc onto the newly formatted disc.

3 Now load in your Basic program.

Type:

```
save "myprog.prg"
```

The name *myprog* can be changed to any eight character string for the filename but the extension of .prg must be included. STOS will now ask you to insert a disc containing the STOS folder, into drive A. This, of course, is the disc which has the system files copied onto it by STOSCOPY.

4 STOS saves out your program in a special format so that it now becomes a proper .PRG file, executable from Gem.

5 If you want your file to auto boot – in other words load when you switch on the computer – you must create a folder called AUTO. You then copy your file into the AUTO folder and whenever you insert this disc into drive A and turn on the ST, your program will automatically load and run.

Commercial STOS Programs

When a runtime file has been generated, it still requires protecting if it is to be released commercially – otherwise you'll be giving away a complete copy of STOS Basic at the same time. On the STOS Basic disc is a file called PROTECT.BAS, this is used to save out a special version of the Basic which does not include the editors commands – which means that other ST owners cannot change your program or write their own STOS Basic programs by typing NEW.

The three main rules for STOS programs which are to be commercially released are:

- You must protect all programs using the PROTECT.BAS program.
- The program must state that it was was written in STOS Basic. A specially-designed sprite with the STOS logo can be found in the SPRDEMO.MBK file

and a STOS icon logo is available in the ICONS.MBK file. You could also use the picture files from within the STOS folder.

- The program must be your own work and not copied in part or whole from the Basic files enclosed on the Accessories and Games discs. No royalty is payable to Mandarin Software – so you are free to do what you like with any games you write.

Adding a title screen

A runtime file searches the STOS folder for a degas picture file – called pic.pi1 or pic.pi3 when it boots up. If it finds the required file it will spin it onto the screen in the same fashion that STOS Basic does when it loads its own title page. This gives your program a professional look and something to display while it loads up all the system files.

Running other files

ONCE the runtime copy of your program has loaded it can run any other Basic program with the command:

```
run "demo.bas"
```

The file *demo.bas* will then be loaded into memory and run.

THE following file would set MODE 0 and then load up the Sprite editor.

```
10 fade 3 : wait 21: mode 0 : run "sprite.bas"
```

Of course you must save sprite.bas onto the same disc and make sure it's a .bas file. Using this technique you can generate integrated suites of programs.

Send it to Mandarin

Mandarin Software are always looking out for new and exciting programs, so if you develop an original, top quality product – or have any interesting ideas – we will be pleased to hear from you. Send your disc with a stamped addressed envelope to:

The Software Manager, Mandarin Software, Europa House, Adlington Park, Adlington, Macclesfield SK10 4NP.

Appendix C

The STOS Basic floppy discs

We have included three single density discs in the STOS Basic package, each of which hold vital data, from the Basic language itself to a space shoot-'em-up game. We were not able to finalise the running order for the discs by the time this manual went to press, so you may find that some of files may not be on the disc as specified below but they will be on one of the discs. You may even find that there are additional files on the discs for your use.

Disc 1 (STOS Basic system disc)

This is the most important of all the three discs and must be backed-up (see Chapter 1). On this disc lies all the system files that STOS loads up, and if various files get deleted then your STOS Basic won't be able to function. The list below explains what each file is for and informs you if files can be changed to your liking.

BASIC.PRG
Double clicking on this file will take you into STOS Basic from the Gem Desktop.

PROTECT.BAS
This program protects run-time programs for commercial release by removing the editor from the copy of STOS Basic it saves to disc. (see Appendix B).

CONFIG.BAS
Use this program to set up the system defaults which dictates the environment that STOS Basic boots-up into.

FOLDER 1 : AUTO *(Runs STOS on boot up)*

START.PRG
This file loads up STOS when the system is booted from a complete reset.

FOLDER 2 : STOS *(Holds all the system files)*

There are various files included in this folder, many of which are vital to STOS. It's best if you don't store any files in the folder – just keep it as it is.

The files in the STOS folder can be split into categories. The main belt of files are the .BIN files which contain the code that the functions from STOS call.

BASIC.BIN
Contains all the control code that makes STOS operate.

FLOAT.BIN
The floating point maths functions. This file can actually be deleted or simply stored in another folder if you only want to use integer values. Doing so releases 15K of memory. See Chapter 3 on variables.

SPRITES.BIN
Code to control the sprites

MUSIC.BIN
Code for the music instructions.

WINDOW.BIN
Code for the window manager routines.

RUN.BIN
The data in this file supplies STOS with the necessary code to allow runtime files to be saved. If you remove this file from the STOS folder you will be unable to save .PRG files.

COMPACT.EXA
This is not a .BIN file but something very similar – an extension file. Extension files are picked up by STOS and the new commands in the file are added to the existing list. This file holds the commands for compacting and uncompacting screens.

The next files are environment files which can be altered to suit your needs.

8X16.CR2, **8X8.CR0** and **8X8.CR1**
These three files are the system character sets that are used by STOS when it boots up. All three files can be altered (see Chapter 8).

MOUSE.SPR
The mouse pointer sprites are held in this file and can also be altered.

PIC.PI1 + PIC.PI3
These are two DEGAS pictures which STOS picks up depending which resolution you are in. The picture is then spun into view and the rest of the STOS system files are loaded in. You can customise your copy of STOS Basic by changing these pictures to whatever you like (See Appendix B for more details).

Disc 2 (Accessories disc)

On this disc are various accessory files which can be used in conjunction with STOS Basic and the program that you are developing. We have included many such accessories, all of which help speed up program development.

Some of the accessories load and save data, in these cases we have included example files to show what can be accomplished with these particular accessories.

Here is a list of the files on the disc, explaining the purpose of each program. The accessories have a .ACB extension and any data files for the accessory will be listed below it.

SPRITE.ACB

This is the sprite definer program which allows you to draw graphics sprites for your program. You can load and save data, grab sprites directly from memory or a disc and it is also possible to grab sprites from Neochrome or Degas pictures and even from commercial games.

We have supplied five files which can be loaded into this sprite editor, these are as follows:

ANIMALS.MBK
In this file you will find frames that make up three animated creatures: an octopus, monkey and a dog.

DROID.MBK
This data file contains animation frames for a superbly designed android.

SPRDEMO.MBK
There are various sprites in this file and you are welcome to use them in your own programs. This file includes a STOS Basic logo which we would like you to include on the title page of your programs.

BACK.MBK
The sprites in this file are to be used in the MAP.ACB accessory but can be edited in the sprite editor.

FONTSET.MBK
In this file there is a font of large characters that can be printed out and animated using the sprite commands.

SPRITE2.ACB

The file SPRITE.ACB is designed to work in low resolution only, thus programs that work in medium and high resolution cannot use it. We have therefore supplied a version which works in all three modes. The only three files that you can load into this version are SPRDEMO.MBK, FONTSET.MBK and BACK.MBK.

MUSIC.ACB

This accessory allows you to develop tunes which you can incorporate in your programs.

MUSIC.MBK
Contains an example of music created by the MUSIC.ACB accessory.

FONT.ACB

An accessory which can be used to create character sets.

FONT1.MBK, FONT2.MBK and FONT3.MBK
These three files are fonts which have been created using the font accessory. Please feel free to use them in your programs.

ICON.ACB

Another accessory which allows you to create images – in this case it gives you the power to create icons.

ICON.MBK
This is an example file created from ICON.ACB and you are free to use any of them in your programs.

COMPACT.ACB

Whole or parts of a screen can be compacted into a special format using this accessory.

BACKGRND.MBK
An example of a compacted screen. See UNPACK for more details on how to unpack this file.

MAP.ACB

Information for map-based games can be generated with this program. Sprite data can be loaded or grabbed and then used to represent the various blocks that make up a background map.

MAP.MBK
This file contains a ready made map and can be loaded into the MAP.ACB accessory. The sprites for the map are contained in the file BACK.MBK.

DUMP.ACB

With this accessory you can dump out the contents of program's memory banks. It lists it in hexadecimal notation and as ASCII characters.

TYPE.ACB

A file can be loaded in and printed to the screen or printer with this accessory. The incoming data is not formatted in any way.

MOUSE.ACB

The coordinates of the mouse pointer are reported by this accessory. This enables you to find out the x,y coordinates of various areas of a Neochrome or Degas screen.

ASCII.ACB

A table of the ASCII characters is listed with this accessory file, enabling you to determine codes quickly.

SCANASCI.ACB

Keycodes and key scancodes can be found using this small but useful utility.

Disc 3 (Games disc)

The Games disc contains three folders, each of which contain a Basic game. These games are:

BULLET TRAIN

In this game you guide a train along a series of tracks avoiding dead end junctions and blasting rail trucks out of your path. The game shows off just how fast STOS Basic can be made to run with the super-fast horizontal scrolling, coordinated animation and fantastic sound.

ORBIT

Another example which displays STOS in all its true colours. Quick reactions are required to play this highly skilled game. Not only do you have 20 challenging levels to play but you can also design and add your own screens.

ZOLTAR

The versatility of STOS is really demonstrated in this game. From the user-friendly

menu system to the powerful designer which allows you to create new waves of alien attack patterns.

To run the above games go into STOS Basic and load one in and then type the RUN command. You can also list and edit the programs.

Here is a list of the files on the games disc and a description of what each one is for.

FOLDER 1 : BULLET

BULLET.BAS
This is the BULLET TRAIN Basic file which you must load from STOS Basic if you wish to play it.

FOLDER 2 : ORBIT

ORBIT.BAS
This file is the one you load into STOS when you want to play the Orbit game.

LEVEL1.ORB – LEVEL20.ORB
These are the 20 screens that have been designed for the ORBIT game and you can edit any one of them or even add new screens by running the ORBIT.BAS program and using the built-in editor.

FOLDER 3 : ZOLTAR

ZOLTAR.BAS
Load this file and type RUN to play the ZOLTAR game.

PHASE1.ZOL – PHASE5.ZOL
These files are the five pre-defined levels which can be altered and many more levels can also be added.

All the accessories and games on the three discs are written in STOS Basic – and you will learn a great deal by examining the listings with the help of commands like SEARCH.

Please feel free to modify any of these programs to suit your needs – and either send us or tell us about the finished results. You never know – we may want to incorporate your program in a future release of STOS.

Appendix D

Using Assembly Language

STOS Basic includes many facilities which allow you to combine assembly language routines with your Basic programs. Usually this isn't really necessary, but sometimes a little machine-code can work wonders even in a language as powerful as STOS Basic.

CALL *(Calls a machine-code program)*

CALL address

CALL allows you to execute any assembly language program held in the ST's memory. *address* can be, either the absolute location of your code or the number of one of STOS Basic's 16 memory banks.

Calling a machine-code program

1 Reserve some memory for your routine using RESERVE AS DATA

Example:

```
RESERVE AS DATA 7,10000
```

The above command reserves 10,000 bytes in bank 7 for your routine. Note that this only needs to be done once as these DATA banks are always saved along with your Basic program. Alternatively, you can also place your code in a previously defined string variable, provided it is **completely** relocatable.

2 Load the program using a line like:

```
load "file.prg",7
```

This program **must** be in TOS relocatable format in order to be usable from STOS. Also note that the extension used for the file should always be PRG and that any other extensions will generate an error message. Never try to call a Gem program from STOS Basic or the system will crash completely!

3 Pass any input parameters using the pseudo variables DREG(0)- DREG(7) and AREG(0)-AREG(6)

4 Call your program using a line like:

```
call 7
```

Your assembly language program may subsequently change any 68000 registers it likes with the sole exception of A7, and must always be terminated with an RTS instruction. It must *never* call the Gemdos traps SET BLOCK, MALLOC, MFREE, KEEP PROCESS or any other memory management function.

Machine code control instructions

AREG *(Variable used to pass information to the 68000's address registers)*

AREG(r)

AREG is an array of six PSEUDO variables which are used to hold a copy of the first six of the 68000's address registers. This enables you to pass information to and from a machine code function executed by either the CALL or the TRAP instructions.

r may range from 0-6 and indicates the number of the address register which is stored in the variable.

Whenever the CALL or the TRAP commands are executed, the contents of this array are loaded automatically into address registers A0-A6. At the end of the function call they are loaded back with any new information which has been placed in these registers.

See DREG, TRAP and CALL

DREG *(Variable used to pass information to the 68000's data registers)*

DREG(r)

This is an array of seven elements which hold a copy of the contents of the 68000 data registers. The number *r* refers to the register number and can range from 0-7 for registers D0-D7. See TRAP for an example of this function in action.

TRAP *(Calls a 68000 trap function)*

TRAP n [,parameters]

TRAP allows you to call one of the numerous 68000 TRAP functions. These traps are really just large libraries of assembly language functions which are available from a single machine-code instruction. You can utilise the TRAP command to give you complete control over the inner workings of your STOS Basic programs. However you should remember that you are effectively programming in machine code. This means that if you play around with the TRAP instruction indiscriminately, you will almost certainly CRASH the ST.

n refers to the number of the TRAP and may range from 0 to 15. Not all of the 16 possible TRAPs have been currently installed into the STOS system. Here is a list of the available numbers:

0,1,13,14 (The Gemdos functions)

3,4,5,6,7 (The STOS functions)

A list of the various Gemdos functions can be found in any good book of machine-code programming on the ST.

The optional parameters specify the data which is to be placed on the 68000's stack before the TRAP function is executed. As a default these are assumed to be of size WORD.

You can set the size directly from the TRAP instruction using a statement such as:

W,expression (Sets the size to WORD)

L,expression (Sets the size to LONG WORD)

expression can be any list of WORDS or LONG WORDS which need to be loaded onto the stack when the function is called.

One useful bonus is that you can also include a string variable in the expression, such as A$. In this case only the ADDRESS of the string is placed on the stack, and a chr$(0) is automatically added to the end of the variable to convert it into the correct format. Another way of passing information to the TRAP is using the PSEUDO registers AREG and DREG. See the appropriate section on these functions for more details.

Here are a few simple examples of the TRAP function in action.

```
trap 14,33,4:rem Set printer type to EPSON

dreg(0)=44:dreg(1)=100:dreg(2)=100: trap 5:rem Move mouse to 100,100
```

STOS Assembly language Interface

STOS provides a wide variety of powerful facilities for the assembly language programmer. These allow assembly language routines to be directly incorporated into STOS Basic programs. Two sets of STOS functions are included. The first of these is basically an expanded version of Gemdos and uses system TRAP number 4. Unlike Gemdos, any parameters are passed to the TRAP using registers. The function number is placed in register D0 and any other data in registers D1 and A0. After the routine has executed, these registers return the results, if any, of the call. All the other registers are unchanged. Here is a list of the various TRAP 4 routines

$0 SCONIN	Get a character from the keyboard.
Input Parameters	D0 = $0
Output Parameters	Bottom byte of D0.W holds Ascii code of key, Top byte contains SCANCODE
$01 SCONIN with ECHO	Get a character from the keyboard and print it on the screen.
Input Parameters	D0 = $1
Output Parameters	Bottom byte of D0.W holds Ascii code of key, top byte contains SCANCODE
$02 SCONOUT	Prints a character contained in D1 onto the screen
Input Parameters	D0 = $2 D1 = Ascii code of the character to be printed
Output Parameters	NONE

Example:

```
MOVE #2,D0
```

```
MOVE #"B",D1
TRAP #4
RTS
```

This prints a "B" onto the screen

$03 READLINE Reads a a string from the keyboard

Input Parameters
D0 = $3
D1 = Maximum number of characters to be input
A0 = Address of Buffer to hold string

Output Parameters
A0 = Pointer to BUFF

Note that this is almost identical to the READLINE function of Gemdos. Like the Gemdos function CONTROL+C aborts the program.

Example:

```
MOVE #3,D0
LEA LEN(PC),A0
MOVE.B #20,d1
TRAP #4
RTS
LEN: DC.W 0
BUFF: BDF 20,0
```

On return, LEN contains the number of characters in the string.

$04 SPRT Prints out a character in D0 to the printer.

Input Parameters
D0 = $4
D1 = ASCII character

Output Parameters
D0 = 0 if an error has occurred.

$05 SPRINT LINE
Prints a line of text on the screen.
Can use standard escape codes.

Input Parameters
D0 = $5
A0 = Address of string to be printed

Output Parameters
NONE

Note that the string must be terminated by a zero.

Example:

```
LEA ADR(PC),A0
MOVE #5,D0
TRAP #4
ADR: DC.B 27,"STOS",0
```

$06 SPRINT VID
Print a line of text of the screen. This is identical to SPRINT LINE except for the fact that escape codes are not translated.

Input Parameters
D0 = $6
A0 = Address of string to be printed

Output Parameters	NONE
$07 BINHEX	Converts a binary number in D0 to an Hexadecimal string pointed to by A0.
Input Parameters	D0 = $7 D1 = number to be converted
Output Parameters	A0 = Address of hexadecimal string

Example:

```
MOVE #7,D0
MOVE #$FFFFA304,D1
TRAP #4
MOVE #5,D0
TRAP #4
RTS
```

$08 HEXBIN	Converts a Hexadecimal string pointed to by A0 into a binary number returned in D0
Input Parameters	D0 = $8 A0 = Address of hexadecimal string
Output Parameters	D0 = Binary result
$09 BINDEC	Converts a Binary number in D1 into a Decimal string pointed to by A0
Input Parameters	D0 = $9 D1 = number to be converted
Output Parameters	A0 = Address of decimal string
$0A DECBIN	Converts a decimal string pointed to by A0 into a binary number returned in D0
Input Parameters	D0 = $A A0 = Address of decimal string
Output Parameters	D0 = Binary result
$0B UPPER	Converts a string of characters pointed to by A0 into upper case
Input Parameters	D0 = $B A0 = Address of string
Output Parameters	A0 = Address of upper case string
$0C EXIST	Searches the current drive to see if the file name pointed to by A0 is on the disc.
Input Parameters	D0 = $C A0 = Address of filename (terminated by 0).
Output Parameters	D0 = Contains the length of the file, or -1 if file not found

$0F CLS — Clears the ST's screen

Input Parameters — D0 = $F

Output Parameters — NONE

$10 LOCATE — Moves the cursor to desired postion on the screen.

Input Parameters — D0 = $10
D1 = Top half of D1 holds X coord, and bottom half holds Y coord

Output Parameters — D0 = None

Example:

```
MOVE #$10,D0
MOVE #$000A0006,D1
TRAP #4
RTS
```

This positions the cursor at 10,6

$11 BREAK — This function prints out the contents of registers D0-D7 and A0-A6 in hexadecimal

Input Parameters — D0 = $11

Output Parameters — D0 = None

Note D0 is printed out as D0*4 by this function.

Example:

```
MOVE #$11,D0
TRAP #4
MOVE #0,D0
TRAP #4
```

$12 READ — Reads a file from the disk

Input Parameters — D0 = $12
A0 = Pointer to Parameter Block
Parameter Block = Pointer to input BUFFER filename

Output Parameters — D0 = -1 if the file does not exist

Example:

```
MOVE #$12,D0
LEA ADR(PC),A0
TRAP #4
RTS
ADR: DC.L STOCK
DC.B "FILE.DAT",0
STOCK: BDF 1000,0
```

$13 WRITE — Writes a file to the disc

Input Parameters — D0 = $13
D1 = No of bytes to be written
A0 = Pointer to Parameter Block
Parameter Block = Pointer to input BUFFER filename

Output Parameters — D0 = -1 if the file does not exist

Example:

```
MOVE #$13,D0
MOVE.L #10,D1
LEA ADR(PC),A0
TRAP #4
RTS
ADR: DC.L BUFF
DC.B "TEST.DAT",0
BUFF: DC.B "ABCDE12345"
```

$14 CHDRIVE — Change the current drive

Input Parameters — D0 = $14
D1 = Drive no (0 .. 3)

Output Parameters — D0 = NONE

$15 CHDIR — Change the current directory

Input Parameters — D0 = $15
A0 = pointer to string containing the pathname

Output Parameters — D0 = NONE

$16 MKDIR — Install a new subdirectory on the disc

Input Parameters — D0 = $16
A0 = pointer to string containing the new directory name

Output Parameters — D0 = NONE

$17 RMDIR — Delete a subdirectory

Input Parameters — D0 = $17
A0 = pointer to string containing the name of the directory to be erased.

Output Parameters — D0 = NONE

$18 KILL — Erases a file or group of files from the disc

Input Parameters — D0 = $18
A0 = pointer to string containing the name or the pathname of the file(s) to be erased.

Output Parameters — D0 = NONE

$19 ASCII

Dumps a buffer containing ASCII text to the printer. Only bytes between $20 and $7F are printed out. Any other characters are replaced by a "."

Input Parameters

D0 = $19 D1 = number of bytes to be printed
A0 = Address of print buffer

Output Parameters

D0 = NONE

Example:

```
MOVE #$19,D0
MOVE #512,D1
LEA BUF(PC),A0
TRAP #4
RTS
BUF: BUFFER
```

$1A FLOPR

Reads one or more sectors from the disc

Input Parameters

D0 = $19
D1 = Read parameters. Lowest word contains the starting sector, the next byte holds the number of sectors to be read, and the top byte of D1 is set to the drive number (0,1,2)
A0 = Data Buffer

Output Parameters

D0 = NONE

Example:

```
MOVE #$1A,D0
MOVE.L #$0001000B,D1
LEA BUF(PC),A0
TRAP #4
RTS BUF:
BDF 1000,0
```

$1B FLOPW

Writes one or more sectors to the disc. parameters identical to the above call, except that D0 contains function no $18.

$1C MUL32

Multiply two 32 bit numbers together

Input Parameters

D0 = $1C
A0 = Address of a buffer area containing 1 long word for the result, and 2 long words holding the 2 numbers to be multiplied.

Output Parameters

D0 = Result of calculation.

Example:

```
MOVE #$1C,D0
LEA R(PC),A0
TRAP #4
 RTS
R: DC.L 0
DC.L $A0000,$FF
```

On return both D0 and R contain the result. ($09F60000 in the example above)

$1D DIV32 — 32 by 32 bit division.

Input Parameters — D0 = $1D
A0 = pointer to a buffer containing 5 long words.

LONG WORD — 1 = 0
LONG WORD — 2 = DIVIDEND
LONG WORD — 3 = DIVISOR
LONG WORD — 4 = 0
LONG WORD — 5 = 0

Output Parameters — D0 = 0 if an error has occurred, non zero if no error.
D1 = Result
A0 = pointer to 2 long words containing the quotient and the remainder of the division.

Example:

```
MOVE #$1D,D0
LEA BUF(PC),A0
TRAP #4
RTS
BUF: DC.L 0
DC.L $FFFFFFFE,2,0,0
```

$1E DIV64 — Performs a 64/32 bit division
Input Parameters — D0 = $1E
A0 = pointer to a buffer containing 5 long words.

LONG WORD — 1 = Bottom half of DIVIDEND
LONG WORD — 2 = Top Half of DIVIDEND
LONG WORD — 3 = DIVISOR
LONG WORD — 4 = 0
LONG WORD — 5 = 0

Output Parameters — D0 = 0 if an error has occurred, non zero if no error
D1 = Result
A0 = pointer to 2 long words containing the quotient, and one long word holding the remainder of the division.

$FFFF SET USER — Install a user defined function.

Input Parameters — D0 = $FFFF
A0 = Address of the start of the new routine

Output Parameters — D0 = NONE.

Example:

```
MOVE #-1,D0
LEA USR,A0
TRAP #4
RTS
```

```
USR: MOVE #0,-(SP)
User function
MOVE D0,D3
RTS
```

$1F USER	Calls the user function defined by SET USER
Input Parameters	D0 = $1F
Output Parameters	Up to you.

Appendix E

The STOS Basic Traps

STOS Basic was written in a very modular way. Each separate group of Basic functions was implemented using a special set of 68000 TRAPs, placed on the STOS system disc. The Traps can be found in the files:

WINDOWS.BIN (TRAP #3)

SPRITES.BIN (TRAP #5)

FLOAT.BIN (TRAP #6)

MUSIC.BIN (TRAP #7)

These files are installed by STOS Basic into memory whenever it is loaded. The advantage of this approach is to allow the machine code programmer unprecedent access to the heart of the STOS Basic system. You can call up most of the more interesting features of the package such as sprites or music directly from assembly language. You should be very careful when using these functions as it's quite easy to make a serious mistake and crash the system. Also note that it's good practice to avoid accessing a function from machine code at the same time as it is being utilised by the Basic as this can lead to unforseen errors.

The window functions (Trap 3)

TRAP 3 supports a list of TRAP functions which make it very easy to create and manipulate STOS windows from within an assembly language program. Instead of using the stack, these routines require all their information to be placed in one or other of the 68000's registers. The function number is stored in register D7 and any additional data is loaded into D0-D1 and A0. If the function returns any results, these will be passed to your program in either A0 and D0. **Warning!** Some of these functions automatically redraw all the sprites on the ST's screen! You can avoid this by using the UPDATE OFF command from Basic.

Here is a list of the various functions:

No.	Name	Action	Parameters
0	CHROUT	Print a character in current window	D0=Character to be output
1	PRINT STRING	Prints a string of characters in window	A0=Pointer to string String is terminated by 0
2	LOCATE	Move text cursor	D0=X coordinate (TEXT) D1=Y coordinate. See LOCATE
3	SET PAPER	Set paper colour	D0=Colour index of paper
4	SET PEN	Set text colour	D0=Colour number of pen

5	TEST SCREEN	Find character under cursor	Returns with character in D0
6	INIT WINDOW	Initialize a window	
7	STOP INTER	Stop interrupts used by windows	DO NOT CALL
8	WINDON	Activate window	D0=Window number
9	DEL WINDOW	Delete window	D0=Window number
10	INIT MODE	Initialise a screen in a new resolution	
11	GET BUFFER	Get address of keyboard buffer	D0=Length A0=Address
12	WINDCOPY	Print current window on printer	
13	GET CURRENT	Get current window no	Returned in D0
14	FIX CURSOR	Change size of cursor	D0=Top D1=Bottom D2=0
15	START INTER	Start window interrupts	DO NOT USE
16	QWINDOW	Activate window quickly	D0=Window number
17	GET CURSOR	Get position of text cursor	Returns Top byte of D0=X coordinate Bottom byte of D0=Y
18	CENTRE	Prints centred text string on the screen	A0=Address of string to be printed
19	SET BACK	Change address of sprite Background	A0=Address of new Background
20	AUTO INS	Opens a space in the current line and places a character in it	D0=Character to be output
21	JOIN	Joins current line with following line	
22	SMALL CURSOR	Displays a small cursor	
23	TALL CURSOR	Displays a thick cursor	
24	MOVE WINDOW	Move a window to new position	D0=Window number D1=X coord, D2=Y coord (Text)
26	SET ICON ADR	Set address of ICONS	A0=Address of ICON BANK

28	GET CHARSET	Get address of character set	D0=Set number Returns address in D0
29	SET CHARSET	Set new address of character set	D0=Set number A0=Address of new set
30	BORDER	Change the border of	D0=New Border (0-16) the current window
31	TITLE	Add a title to the current window	A0=Address of a string for title (terminated with 0)
32	AUTOBACK ON	Identical to Basic version.	
33	AUTOBACK OFF	See Basic version	
35	XGRAPHIC	Convert X coord from text to graphic	D0=Text coord Returns converted coord in D0
36	YGRAPHIC	Convert Y coord from text to graphic	D0=Text coord Returns converted coord in D0
37	XTEXT	Converts X coord from graphic to text	D0=Graphic coord Returns converted coord in D0
38	YTEXT	Converts Y coord from graphic to text	D0=Graphic coord Returns converted coord in D0
39	SQUARE	Draws a square at current cursor position	D0=Border (0-16) D1=Width (Minimum 3) D2=Height (Minimum 3)

The sprite functions (Trap 5)

The STOS Basic sprite commands are performed using a special section of the STOS system called the SPRITE MANAGER. This handles all the interrupt-driven movements and animations which make STOS Basic so amazingly powerful. You can communicate with this process from machine code by using a simple set of TRAP 5 instructions. These take the function number in register D0, and read the various parameters in the other registers. Note that only registers D0-D1 and A0 are modified by this TRAP.

No	Name	Action	Parameters
1	INIT MODE	Initialise the sprite generator to a new resolution	
2	CHANGE BANK	Change the address of the sprite bank. See Pn for more details	A0=Address of new sprite bank
3	CHANGE LIMITS	Change limits of the display area used by the sprites. (Called by LIMIT SPRITE)	D1=X Coord of Leftmost limit D2=X Coord of Right limit D3=Y Coord of Top limit D4=Y Coord of Bottom limit

4	SYNCHRO	Turns on/off synchro-nisation of sprites and background (See SYNCHRO from Basic)	D1=1 for SYNCHRO ON D1=0 fro SYNCHRO OFF
5	PRIORITY	Switch between normal & Y coordinate priority (See PRIORITY from Basic)	D1=1 for PRIORITY ON D1=0 for PRIORITY OFF
6	POS SPRITE	Get position of sprite Returns X coord in D0 and Y coord in D1	D1=Sprite number
7	SPRITES ON/OFF	Redraws or remove all sprites on screen	D2=1 for Redraw D2=0 for erase
8	SPRITE ON/OFF	Redraws or removes one sprite on screen	D2=1 for Redraw D2=0 for erase D1=Number of Sprite
9	SPRITE	Draws a sprite	D1=Number of sprite D2=X coordinate of sprite D3=Y coordinate of sprite D4=Image number of sprite
10	MOVES ON/OFF	Starts or stops all sprite movements	D2=0 for STOP D2=1 for FREEZE D2=2 for START
11	MOVE ON/ OFF	Starts or stops one sprite movement	D2=0 for STOP D2=1 for FREEZE D2=2 for START D1=No of sprite
12	MOVE INIT	Defines a sprite movement Eqivalent to MOVE X and MOVE Y	A0=Address of movement string terminated by a zero (in same format as Basic) D1=No of sprite D2=0 for MOVE X D2=1 for MOVE Y
13	ANIMS ON/OFF	Same as function 10 for animations	
14	ANIM ON/OFF	Same as function 11 but for animations	
15	INIT ANIM	Define an animation sequence.	A0=Address of animation string terminated by 0 (in same format as Basic) D1=Number of sprite
16	UPDATE	Redraw any sprites which have changed since last update	

17	SHOW	Show mouse	D1=0 for SHOW ON D1=1 for SHOW
18	HIDE	Hide mouse	D1=0 for HIDE ON D1=1 for HIDE
19	CHANGE MOUSE	Changes mouse image	D1=No of new image
20	MOUSE	Get mouse coordinates	Returns X coord in D0 Y coord in D1
21	MOUSEKEY	Get mouse button returns	Returns status in D0
22	SCREEN TO BACK	Copies physical screen to sprite background	
23	BACK TO SCREEN	Copies sprite background to physical screen	
24	DRAW MOUSE	Redraw mouse on screen	
25	SET ZONE	Set test zone	D1=No of zone D2=Leftmost limit in X D3=Rightmost limit in X D4=Top limit in Y D5=Bottom limit in Y
26	ZONE	Test zone	D1=Sprite to be tested Returns zone number it was found in or 0 in D0
27	CHANGE BACK	Change address of sprite background	A0=New address
28	STOP MOUSE	Stop the mouse moving on the screen	
29	DRAW SPRITES	Redraws all the sprites on the screen	
30	START INTER	Starts sprite interrupts	DO NOT USE!
31	STOP INTER	Stops sprite interrupts	NEVER USE THIS FUNCTION!
32	LIMIT MOUSE	Limit mouse to area on screen	D1=X coord of Left corner D2=Y coord of Left corner D3=X coord of Right corner D4=Y coord of Right corner
33	SCREEN COPY	As STOS Basic	A0=Address of source screen A1=Address of dest screen D1/D2=(X,Y) of rectangle to be copied D3/D4 (X,Y) of destination D5/D6 (W,H) of zone to be copied.

34	ICON	Put Sprite	D1=X coord of sprite D2=Y coord of sprite D3=No of Icon address of sprite data
35	PUT SPRITE	Puts Sprite in background screen, providing it is already displayed	D1=Number of sprite
36	INIT ZONE	Initialise test zones	
37	GET SPRITE	Equivalent to the Basic instruction	D1=X coordinate of new sprite D2=Y coordinate D3=Pointer to sprite to be copied. D4=Mask
38	REDUCE	Reduce a screen	A0=Address of source screen A1=Address of destination D1=X coord of reduced screen D2=Y coord of reduced screen D3=Width of reduced screen D4=Height of reduced screen
39	INIT FLASH	Initialise colour flashes	
40	FLASH	Set up a flash sequence	D1=No of colour to be flashed A0=Flash string terminated by a zero. See FLASH from Basic
42	ZOOM	Enlarges a section of the screen	A0=Address of source screen A1=Address of destination D1=X coord of top left corner D2=Y coord of top left corner D3=Width of the section D4=Length of the section D5/D6=Coordinates of dest A2/A3=Size in X and Y of dest
43	APPEAR	Fades between two screens	A0=Address of source screen A1=Address of dest screen D1=Type of fade (1-80)
44	MOVE MOUSE	Changes the coordinatesof the mouse	D1=New X coordinate D2=New Y coordinate
45	MOVON	Checks whether sprite is in motion	D1=No of sprite Returns 0 in D1 if sprite is not moving and 1 if the opposite is true
46	SHIFT	Shifts the palette of colours.	D1=Speed in 50ths of a second D2=Colour the rotation is to be started at.
47	REDRAW	Identical to the Basic function.	

Floating point extension library

This gives the programmer access to a wide variety of floating point operations and uses numbers in the IEEE 64-bit format between 10 E-307 to 10 E+308. These routines corrupt registers D0-D4 and A0-A1. As before, the function number is loaded into D0 before calling the appropriate routine.

The first parameter should always be placed in registers D1-D2, (with D1 containing the bottom half of the number, and D1 holding the top half. If a second parameter is required, this should be put into registers D3-D4 using the same format. You can now execute the function using a TRAP #6 instruction.

$00 ADFL Adds two floating point numbers together

Example:

```
MOVE #0,D0
MOVE.L #$3FF19999,D1          ; First no in D1-D2
MOVE.L #$9999999A,D2
MOVE.L D1,D3                  ; Copy 1st no into
MOVE.L D3,D4                  ; 2nd no
TRAP #6
RTS
```

On return D0.L and D1.L contain the result.

$01 SBFL Subtract one floating point number from another Parameters used identical to ADFL

$02 MLFL Multiply two floating point numbers

$03 DVFL Divide two floating point numbers

$04 SINFL Takes the SIN of the number in D1-D2 and places it in D0-D1.

$05 COSFL Takes the COS of the number in D1-D2 and places it in D0-D1.

$06 TANFL Takes the TAN of the number in D1-D2 and places it in D0-D1.

$07 EXPFL Takes the Exponential of the number in D1-D2 and places it in D0-D1.

$08 LOGFL Calculates the naperien log of the number in D1-D2 and returns the result in D0-D1

$09 LOG10FL Calculates the base 10 log of the number in D1-D2 and returns the result in D0-D1

$0A SQRFL Takes a number in D1-D2 and returns the square of it in D0,D1

$0B ATOFL Takes an Ascii string pointed to by A0 and converts it into a number in floating point format in D0-D1

$0C FLTOA Takes an FP number in D1-D2 and converts it into an Ascii string

Input Parameters

D0 = $0C
D1-D2 = The FP number to be converted.
D3 = A digit representing the number of digits after the decimal point in Ascii.
A0 = The pointer to a buffer for the string

Output Parameters

The length of the Ascii string (not including the final 0)
A0 = A pointer to the string of Ascii characters terminated by a 0.

Example:

```
MOVE.L      #$3FF19999,D1     ; Load 1.1 into D1-D2
MOVE.L      #99999999A,D2
MOVE        #$C,D0
LEA         BUF(PC),A0
MOVE.W      #$0031,D3         ; 1 Digit after the DP
TRAP #6
MOVE        #5,D0             ; Print the number on the
TRAP        #4                ; screen.
RTS
BUF: BDF 1000,0
```

$0D FLTOIN — Convert a FP number in D1-D2 into an integer in D0

$0E INTOFL — Convert an integer in D1 into an FP no in D0-D1

$09 EQFL — Compares the two numbers in D1-D2 and D3-D4. If they are equal then D0 contains a 1, otherwise it contains a zero.

$10 NEFL — Compares the two numbers in D1-D2 and D3-D4. If they are not equal then D0 contains a 1, otherwise it contains a zero.

$11 GTFL — Compare two numbers and return a 1 in D0 if the first is greater than the second.

$12 GEFL — Test if greater than or equal

$13 LTFL — Test if less than

$14 LEFL — Test if less than or equal

$15 ASINFL — Calculate the Arc Sin of no in D1-D2 and return it in D0-D1

$16 ACOSFL — Calculate the arc cos

$17 ATANFL — Calculate the arc tan

$18 SINHFL — Calculate the hyperbolic sin

$19 COSFL — Calculate the hyperbolic cos

$1A TANFL — Calculate the hyperbolic tan

$1B INTFL — Get the integer part of D1-D2 and place the result in D0-D1

$1C POWFL Calculate X^Y where X is in D1-D2 and Y is in D3-D4. As usual the result is in D0-D1

The music generator

Like the sprite definer, there is also a special music generator which functions completely independently of the rest of STOS Basic. This can be called from any of your machine code programs by using a TRAP 7 instruction. To access these routines, place the function number in D0. Note that only registers D0 and A0 are modified by these commands.

The music Traps (Trap #7)

No.	Name	Action	Parameters
0	INIT SOUND	Resets sound generator and kills music	
1	START MUSIC	Starts playing some music	A0=Address of music
2	STOP VOICE	Stops the music played on a single voice	D1=Number of voice
3	RESTART VOICE	Resumes playing a single voice stopped by STOP VOICE	D1=Number of voice voice
4	FREEZE	Freezes some music	
5	UNFREEZE	Resumes some music frozen with FREEZE	
6	CHANGE TEMPO	Change speed of music	D1=New speed (0-100)
7	START INTER	Start music interrupts	DO NOT USE
8	STOP INTER	Stop music interrupts	DO NOT USE
9	TRANSPOSE	Change pitch of music by a number of semi tones	D1=Number of semi tones
10	GET VOICE	Get position of in a voicwe	D1=Number of voice Returns position in D0

PSG *(Access Programmable sound generator)*

PSG(r)

The Atari ST incorporates a special piece of circuitry which it uses to generate the wide range of different sounds which can be played through your monitor or television set. This circuit is built around a single microchip known as the YAMAHA YM 2149. It possesses the following general characteristics.

- 3 separate frequency generators (One for each VOICE)
- 1 noise generator (Used by STOS Basic's NOISE command)
- 15 different volume levels (See VOLUME)
- 16 preprogrammed envelopes (Accessed by ENVEL)

The precise sound produced by the circuit is determined by the contents of 14 different SOUND REGISTERS numbered from 0-13. You can access these registers directly using the PSG command. PSG is effectively an array which holds a copy of the current contents of the sound registers. Whenever you assign a value to one of the elements in the PSG array, this will be automatically loaded into the appropriate register.

Example:

```
print psg(1)
```

WARNING: This function is DANGEROUS! Incorrect usage can cause serious damage to any disc in the current drive. This is because part of the sound chip is also utilised by the ST's disc system. You should therefore take extreme care when attempting to use this command.

Here is a brief list of the various sound registers and their uses.

Register	Function
0	Bits 0-7 set the pitch in units of a single step for voice 1.
1	Bits 0-3 set the size of each frequency step.
2	Fine control for voice 2. Format as Register 0
3	Coarse control for voice 2. As register 1
4	Controls pitch of voice 3 in the same fashion as register 0
5	Coarse control of the pitch of voice 3
6	Bits 0-4 control the pitch of the noise generator. The higher the value the lower the tone.
7	Control register for sound chip. Bit 0: Play pure note on voice 1 ON/OFF (1 for ON, 0 for OFF) Bit 1: Voice 2 tone ON/OFF Bit 2: Voice 3 tone ON/OFF Bit 3: Play NOISE on voice 1 (1 for ON, 0 for OFF) Bit 4: Voice 2 noise ON/OFF Bit 5: Voice 3 noise ON/OFF
8	Bits 0-3 control volume of voice 1. If bit 4 is set to one then the envelope generator is being used, and the volume bits are ignored. Since this corresponds to a volume of 16, this explains why you need to set VOLUME to 16 before you can use the ENVEL command.
9	As Register 8 but for Voice 2
10	As Register 9 but for Voice 3
11	Bits 0-8 provide fine control of the length of the envelope
12	This register provides coarse control of the length of the envelope
13	Bits 0-3 choose which of the 16 possible envelope types is to be used.

Appendix F

Structure of the sprite bank

All of the STOS Basic sprites are stored in bank number 1. It begins with a block of general information about the sprites. This designates the number of sprites in each resolution and their position in memory relative to the start of bank 1.

Offset from start of sprite bank	Meaning
0	Sprite identification code $19861987
4	4-byte offset to address of sprite parameter block in low resolution
8	4-byte offset to address of sprite parameter block in medium resolution
12	4-byte offset to address of sprite parameter block in high resolution
16	Number of sprites in low resolution
18	Number of sprites in medium resolution
20	Number of sprites in high resolution

After this section comes a list of special SPRITE PARAMETER BLOCKS. These hold specific information about each individual sprite and are 8 bytes in length.

Typical sprite parameter block

Offset from start of sprite bank	Sprite 1 parameter block
22	4-byte offset to sprite 1 data
26	Width of sprite 1 (in units of 16)
27	Height of sprite 1
28	X Coordinate of hotspot
29	Y Coordinate of hotspot
30	**Sprite 2 parameter block...**

Finally comes the data which makes up the actual design of the sprites.

Here is a diagram which illustrates its structure.

The Sprite Data Block

Data for Mask (one bit plane)

Sprite Data (organised in Bit Planes)

Although these sprites may look rather complicated, remember that you can design and manipulate STOS Basic sprites without ever needing to know anything about how they are really stored in memory.

Structure of the icon bank

All STOS Basic icons are stored in bank number 2 using the following format:

Offset from start of bank 2	Meaning
0	$28091960 This is the icon bank ID number
4	Number of icons in bank
6	Start of data for icon 1. This is 84 bytes long, and uses the same format as the LINEA sprites.
92	Start of data for icon 2
166	Start of data for icon 3

Structure of the music bank

STOS Basic places all its music data in Bank number 3. Here are full details of how this information is stored in the ST's memory.

Offset from start of Music Bank	Meaning
0	$13490157 This is the identification code used to indicate a Music bank
4	Offset from start of the bank to music number 1 Set to zero if no music with this number
8	Offset to music number 2
124	Offset to music number 32. (Maximum of 32 pieces of music)
128	Length of this memory bank.
132	Name of Music 1 (8 letters)
140	Name of Music 2 (8 letters)
380	Name of Music 32

388	Start of Music 1
388+Length Music 1	Start of Music 2
etc	

Inside the music definitions

Each piece of music starts off with its own individual header block. This contains the definitions of all the envelopes and tremolos you have used, along with information about the position of the various voices which make up the music.

Music Header

Byte Number	Contents	
0	$19631969	This is the Identification code used to indicate that the data is music.
6		Offset to Music in voice 1
8		Offset to Music in voice 2
10		Offset to Music in voice 3
12		Definition of first tremolo/envelope (36 bytes long)
48		Definition of second tremolo/envelope

Start of voice 1

The Music commands

Each note is stored as a two-byte word ranging from 0-32767. The lower half of this word contains the pitch of the note (0-96). See PLAY for more details. The upper byte holds the length of the note in 50ths of a second. The Music commands are held in either two or four bytes. In order to distinguish them from normal notes, the highest bit of these commands is set to 1. Here is a list of the various commands and the numbers used to represent them in the music.

Number	Size	Command	Meaning
$8000	2 bytes	END	Signifies end of music for this voice
$A000	2 bytes	MUSIC	Uses pure tones for music
$A100	2 bytes	NOISE ONLY	Uses noise for music
$A200	2 bytes	STOP NOISE	Turns off noise
$A3xx	2 bytes	NOISE xx	Plays noise with pitch xx
$A400	2 bytes	STOP NTREMULO	Stop Mixing Tremulo with noise

$A500	2 bytes	STOP ENVEL	Stop using current Envelope
$A600	2 bytes	STOP TREMOLO	Stop using current tremolo
$A7xx	2 bytes	VOLUME xx	Set volume of sound to xx
$C000	4 bytes	NTREMULO	Mix TREMULO with noise. Bytes 23 hold offset to tremulo definition
$C100	4 bytes	ENVEL xx	Use ENVEL xx. Bytes 23 hold offset to envelope definition.
$C200	4 bytes	TREMULO xx	Use TREMULO xx. Bytes 2-3 hold offset to tremulo definition
$C3nn	4 bytes	REPEAT nn,note	Repeat music starting from note, nn times. Note held in bytes 2-3.

Screen banks

The format of the screen banks is very straightforward indeed. The first 32000 bytes of this memory hold the actual screen data, and the next 16 words from number 32000 to 32032 contain a copy of the colour settings for this screen. Note that the bytes from 32032 onwards are free, and can be used for your own purposes.

STOS Basic Commands

Index