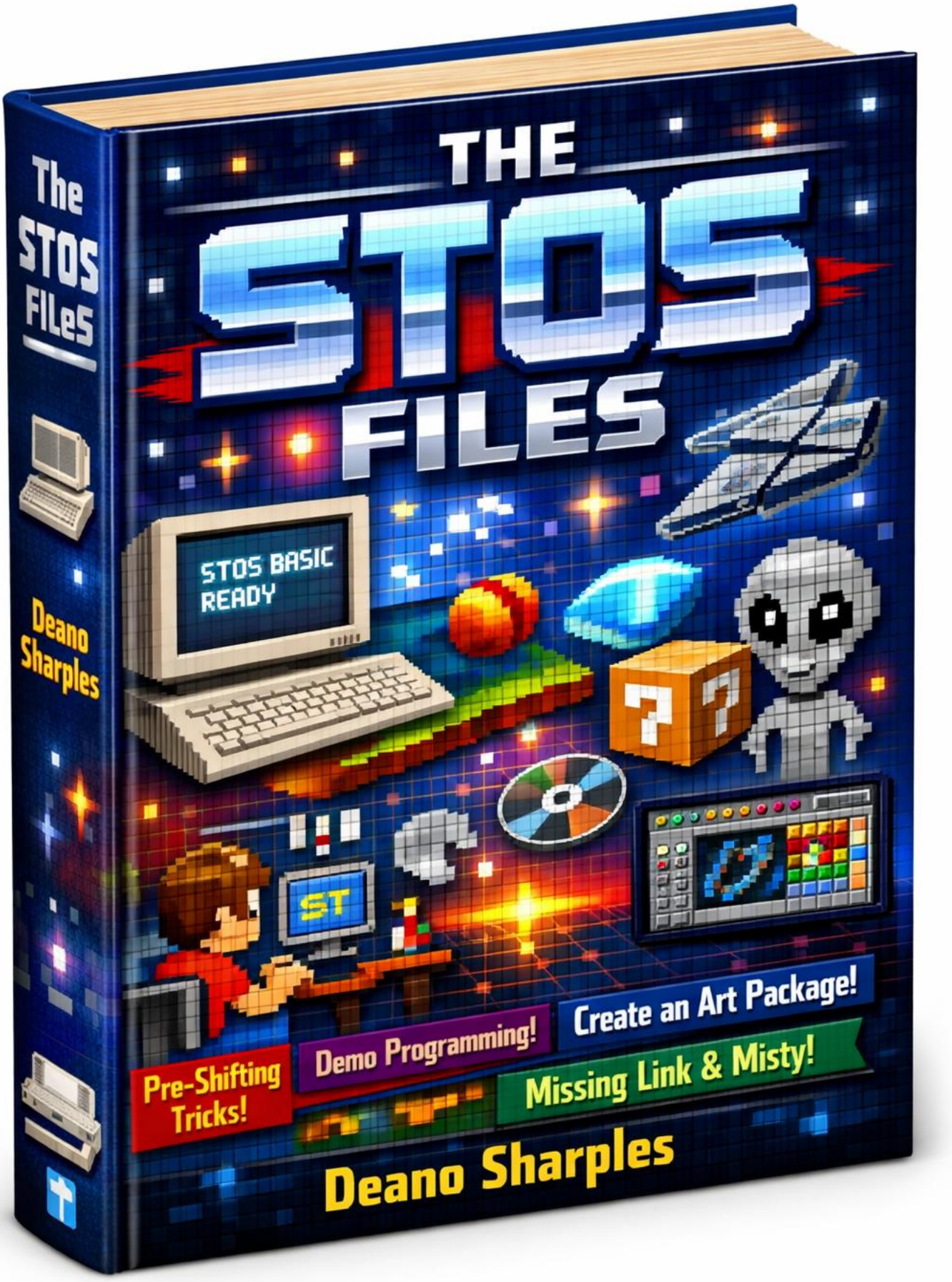
The STOS Files
Deano Sharples
THE STOS FILES
STOS BASIC READY
Pre-Shifting Tricks!
Demo Programming!
Create an Art Package!
Missing Link & Misty!
Deano Sharples

INTRODUCTION

My journey as a STOS programmer started in 1990 when I bought my first and only Atari STE from a computer store in Oldham. It was the turbo pack and came with STOS basic as well as other programs. Eventually I got down to learning how to code and I made a few adventure games. One day I discovered the Stosser Diskzine which not only helped me improve my programming, but I met some great contacts too. I started writing articles and tutorials for Stosser and attempted to have something published in every issue. I became the editor around issue 24 and kept it going till issue 29.

Some time ago I went through all the back issues of Stosser and transferred all my articles and tutorials to my PC, where I painstakingly corrected all my spelling mistakes and punctuation errors and reformatted the documents. After the general cleanup and amendments, I placed them on my website. Now I have decided to put them all in one book for STOS users worldwide to benefit from. Here is a list of articles and tutorials you will find in this book.

CONTENTS

STOS Basic Routines

STOS Code to Amos Code

STOS Compiler Errors

STOS Extensions

Using the STOS Sprite, Move and Anim Commands

Variables and Dimensions

Writing a SAC Adventure Game

This book is more of a reference book than anything else. It is useful if you want a particular tutorial. I have included a bonus article about how to structure your code which you may find useful.

I hope this guide is useful to you in some way. I had some very happy times writing for Stosser and I am happy to present the complete collection of my work for you. Feel free to distribute this book anyway you wish as long as you don’t charge for it.

Deano Sharples

HOW TO STRUCTURE YOUR PROGRAMMING CODE

I remember my first fumble with basic on my ZX Spectrum computer back in the 1980s, ploughing through pages of basic commands and example code without any real idea of how I could write programs myself. It was like reading a dictionary where I could learn certain words and their meanings with limited information on how I could construct them into entire sentences to write a document. Every programmer who has dabbled in basic has probably come across the famous "Hello World" routine which consists of a two-line program that prints this phrase unlimited times on the screen.

Your program code needs to be written as step-by-step instructions using the commands that your choice of programming language understands. It means reading your programming manual to learn which commands you need to use for what you want your program to do. In the "Hello World" example you would first need a command that prints "Hello World" onto the screen, and then you would need a second command to print it again multiple times, without writing multiple print statements.

Check out this example. To make things simple I am using old-school basic with line numbers.

```
10 print "Hello World

20 goto 10
```

The best structure for writing any program code is to make it clear and easy to follow. Some programmers put multiple commands on one line which can make the code difficult to follow if you are trying to iron out bugs. Spreading your code over multiple lines makes the program work better and more readable.

Another recommended practice is to separate each part of your program code using REM Statements. REM (short for Remark) allows you to put comments before each section of code to remind you what each part does. This is especially useful if you wish to edit your code at a later date.

```
10 rem Set Up Variables

20 let A=1 : let B=2

30 rem *******

40 rem Print Variables to Screen

50 rem *******

60 print A,B
```

Anything after the REM command is ignored by the computer. You can use as many REM statements as you want to make bigger gaps in your code for easy reading. Other programming languages allow you to use blank lines or indent the first line of the routine.

Now I will show you how to structure the entire program code. Remember that the computer needs to follow step-by-step instructions so you need to write each instruction in the order you want it to run.

CONSTRUCTION OF CODE

Setup screen resolution and variables: The first section of your program would set the screen resolution and the variables.

Read information into arrays: If you have the information you want to put into an array using the DIM command then you can use a For/Next loop and the READ command. It is best to place the data statements for the array to read from at the end of your program.

Set up the main screen: This is the section where you would use a subroutine (GOSUB Command) to set up the main screen. In a shoot-em-up type game, you would have a routine that draws the sprites and game screen and then returns to the next line of the code it came from.

Main Program Loop: Once the program is up and running the main program loop jumps to various routines using subroutines and then returns to the next line in the loop.

Program Routines: It is a good structure to place all the program routines after the main loop. You would have separate routines that update the screen, check for joystick input, check for collision detection, and so on. After each check, you return to the main loop.

Data Statements: Finally, you can list all the data statements at the end of the program which makes them easier to find and correct if need be.

CONCLUSION

Creating your code with plenty of REM Statements and short lines makes your code look cleaner and easier to follow. There may be a time you want to improve the program or use the routines for other programs.

ADVENTURE GAME PLANNING

Adventure games have been a popular source of amusement for years. This type of game puts you into a created fantasy world where you make the decisions. For example, you could be a knight in shining armour attempting to rescue a princess from a tall tower guarded by a fire-breathing dragon. In adventure games, the player would be told where he is and what's going on around him, with people and creatures to communicate with, and puzzles to solve. A world where your dreams become reality. The first adventure games were 'text based', meaning that you communicated to the game by typing words and it answered back by printing a reply on the screen. For example, typing "get the lamp" will result in the computer replying "You take the lamp" – providing there was a lamp there in the first place. Playing an adventure game is like being a character in a film, what happens depends on your choice.

The only problem with these early adventure games was that they couldn't understand everything the player typed in. So along came the graphic adventure games which hands control over to the mouse with little or no keyboard input. Commands would be clicked on instead of typed. These types of games include the early Monkey Island and Simon the Sorcerer series. These games bring your adventure to life as you can see the world you are in rather than reading a text description.

PLANNING A GAME

Before we can start writing an adventure game, we need to plan it out. First, we need an idea then built the game around it. Let's say our game takes place in a castle. This would be our fantasy game world which is broken down into several small locations. We would then make a list of the various locations you would find in a castle like this:

Throne Room

Main Hall

Drawbridge

Servants Quarters

Bed Chamber

Tower

Dungeon

Castle Grounds

Staircase

The next thing to do is connect each location to the nearest one. For example, the nearest location to the Drawbridge would be the Main Hall or the Castle Grounds. This is important because you don't want the player to enter the castle and find himself stepping into the Dungeon. The list would now look like this.

Location 1: Castle Grounds

Location 2: Drawbridge

Location 3: Main Hall

Location 4: Throne location

Location 5: Dungeon

Location 6: Servant's Quarters

Location 7: Staircase

Location 8: Bed Chamber

Location 9: Tower

OBJECTS

An object is something the player can use in his quest. It could be a coat, a lamp, some food, a bag or anything that can be picked up and moved. Each object has its normal location number, it could also be in special locations which are known as Worn, Carried and Not Created. If the main character is a Barbarian, we could give him a sword and put it in the carried location, so when the game starts the Barbarian is carrying the sword. We could also define a magic potion for him and put it in a normal location waiting for him to find it.

Sword (Carried Location)

Potion (Location Five)

Robe (Worn Location)

If you've ever played an adventure game and found something which was hidden, then you've found a "Not Created" object. This is a normal object without a normal location number. When the player finds the object the object location variable will become the number of the location the player is in.

PUZZLES

In adventure games, puzzles are there to give the player a challenge and make solving his quest harder. For example: finding a way or opening a chest or getting past a guard dog. Let's note down some puzzles for the castle game.

Puzzle: The player has to get past the guard at the drawbridge.

Answer: The player must find a guard's uniform and wear it.

Puzzle: The player can't unlock the door of the Bed Chamber.

Answer: The player must break the lock with his sword.

Puzzle: The player can't get past the guard dog in the Tower.

Answer: The player must find a bone and give it to the dog.

TRAPS

As with puzzles, traps are set up for the player to fall into. If he tried attacking the guard then the guard would kill him and the game would be over. Here are some traps for our castle game.

If the player takes the King's Robe the guards will kill him.

If the player drinks the green potion he will die.

If the player tries to walk past the dog it will bite him.

EVENTS

Events are the outcome of things that happen in the game. For example, if the player had solved the puzzle of the dog, then an event would be called to put the bone in the Not Created location and the player would be told that the dog ran off with the bone. The dog and bone would be removed from the game. Here are some more examples of events

When the player examines the vase, he finds the key.

(Note: put the key in the player's present location).

When the player breaks the lock on the door, he can open it.

(Note: set the DOOR variable to one to indicate it’s open).

When the player drinks the potion, his strength will be restored.

(Note: Set STREN variable to full and put the potion in Not Created location).

EXITS (CONNECTIONS)

The player needs a way of moving between locations so we have to define the exits from each location. Exits take on the form of North, South, East, West, Up, Down, etc. We can also have false exits. This means that the player can’t move to another location just yet, in other words, it’s blocked off. Let’s take our castle game. From the drawbridge we have two exits: one leads to the Main Hall (location 3) and the other leads to the Castle Grounds (location 1). We could make a note like this.

Drawbridge: (North goes to location 3, South goes to location 1)

So, the drawbridge is location 2 and from there the player can go north to location 3 or South to location 1. A false exit could be noted down as an event, so when the player opens the door of the dungeon, he can enter it. (Note: Connect Main Hall to Dungeon). You can think of this as a Not Created location being created.

Before we can get these options working, we need to set up the game data. The best way of doing this is to type it all out as data statements and then read them into an array. So, let’s put our location descriptions down as data statements first.

Data ”You are in a cave, an opening is east.”

Data ”You are outside a cave, the entrance lies west.”

To put and hold this in memory we need to define an array using the DIM command like so.

dim location$(2)

This array will hold two location descriptions, next we need a line to put the location descriptions in the array so we use the READ command.

for X=1 to 2 : read location$(X) : next x

You can check if the line has done its job by typing 'print location$(1)', the description for location one should appear.

Now let's define the EXIT data. For this, we need to use a two-dimension array called MAP.

dim MAP(2,4)

Next, we use a nested loop to read the data into the array.

for X=1 to 2 : for Y=1 to 4 : read MAP(X,Y) : next Y : next X

Finally, the data lines for each location.

data 0,0,0,2

data 0,0,1,0

Here we have a two-dimension array that holds all the exit data for each location. So, in the above example, the first number is the number of locations in the game and the second is the number of exits.

To use this example, we have to give each exit a number for the game to refer to like this.

North – Exit One

South – Exit Two

West – Exit Three

East – Exit Four

So, if we wanted exit two (south) to lead to location 3 we would use the following line of data.

data 0,3,0,0

Here we have four numbers on a line. As the south is exit two then we replace the second number on the line with the number three so we now have an exit leading south to location three. Let's connect a north exit to location four from this location.

data 4,3,0,0

Each data statement must have four numbers on it. If you didn't want to use a certain exit then you would set the exit number to zero.

We also need a routine to allow the player to move from location to location via the defined exits. Use this line.

if MAP(location,CH)<>0 then location=MAP(location,CH) else print”You can’t go that way.”

OBJECTS

We can use the same method to store the object examine messages.

dim OBJECT$(2)for X=1 to 2 : read OBJECT$(X) : next X

data ”a small lamp.”,”a sharp sword.”

OBJECT LOCATIONS

Each object has its own location number which could either be a normal or a special one. First, we set up the array which holds the object location numbers.

dim OB_LOC(4)

Next, we use READ to put it into the array.

for X=1 to 4 : read OB_LOC(X) : next X

And next, the data line containing the object location numbers.

data 1,5,10,15

So, object 1 at location 1, object 2 at location 5, object 3 at location 10, and object 4 at location 15

for x=1 to 4

if OB_LOC(X)=location then print”You can see”;OBJECT$(X)

next X

Where the OBJECT$ array holds the description of the objects IE: a small lamp.

SPECIAL OBJECT LOCATIONS

Normal object locations would be as many as the locations in the adventure game as they appear in the actual game. But there are special object locations such as these.

CARRIED - Object is carried by the player

WORN - Object is worn by the player

NOT CREATED - Object does not yet exist in the game

To keep them separate from normal objects we need to give them higher numbers than the total number of locations in the game.

CARRIED=1000 : WORN=2000 : NC=3000

Each variable can be used to specify which special location the object is in. So, for example, if we wanted the game to list our carried objects then it could be checked like

this. First, using an array like OB_LOC, we can put our first object in the CARRIED special location like this.

OB_LOC(1)=CARRIED

We can then add some lines to our game to inform the player what objects he is carrying……

for X=1 to 10

if OB_LOC(X)=CARRIED then print”You are carrying…..”;OBJECT$(X)

if OB_LOC(X)=WORN then print”You are wearing….”;OBJECT$(X)

next X

I have only covered a few basics here. For full information why not try my STOS Adventure Creator and STOS Graphic Adventure Creator which you can download from my website (deanosharples.com). Most of the work is done for you so you can start knocking out many adventure games.

CREATING PRE-SHIFTED SPRITES

What has always been a pain for ST users is the ST's sixteen-pixel boundary problem. This means that any graphic has to be copied in multiples of sixteen pixels. For example, if you used the command screen copy to put a piece of the picture on screen then it will round its x coordinates to the nearest sixteen pixels. Look at this example.

screen copy 5,10,10,100,100 to physic,10,10

The command would ignore the value of 10 for the x coordinate and place it on screen at x coordinate 16. Always round up to the nearest step of 16, EG: 16,32,48,64, etc. This can be awkward if you need your piece of graphic to be in its exact place.

The only way to overcome this problem is by pre-shifting. This means making several copies of the graphic with a smaller pixel gap in between. Try this example…

```
10 key off : curs off : flash off : hide on : mode 0

20 dim SP$(16)

30 for X=1 to 16

40 sprite 1,X,16,1 : put sprite 1 : wait vbl : sprite off

50 SP$(X)=screen$(logic,0,16 to 16,32) : cls

60 next X

70 rem Show It

80 logic=back

90 for X=1 to 16

100 cls logic

110 screen$(logic,X,16)=SP$(X) : wait 5

120 screen swap : wait vbl

130 next X
```

This routine shows how to make pre-shifted sprites. Lines 10 to 60 make sixteen copies of sprite one and the rest of the routine moves it across the screen at a step of one pixel at a time. Each image of sprite one is caught in the 'screen$' command, each one captured in its own different position moved one pixel across further than the last one. Might you, sixteen images in memory can cost a lot of memory so we can make fewer images, this means the pre-shifted graphic moves in bigger steps but takes up less memory. The pre-shifted graphic can move in either 1,2,4,8, or 16 pixels across the screen. In other words, we are cutting down the 16-pixel box into smaller equal size boxes. We can work out how many images we need and how many steps we need to move in like this.

16/STEPS=?

So, we take the number 16 which is the boundary and we divide it by the number of steps or pixels we want the pre-shifted graphic to move and we get the answer to how many copies we need to make of it. For example, if you wanted to move the sprite across the screen at four pixels at a time then you could try this sum.

16/4=4

This means you would have to make four copies of the sprite. Try other sums but remember 'STEPS' must be either 1,2,4,8, or 16.

Now let's imagine we have a block of a picture 100×100 pixels in size and we wanted to place it at coordinates 36,0 on the screen. Using screen copy would place the block at 32,0. Therefore we need to screen copy the block to a dummy screen at coordinates 32,0 and scroll it along to 36,0. To do this we need to use the Def Scroll command like this.

```
10 key off : curs off : hide on : flash off : mode 0

20 reserve as screen 5 : rem Dummy screen

30 screen copy 6,0,0,100,100 to 5,32,0 : rem copy block from bank 6

40 def scroll 1,32,0 to 132,100,4,0 : rem Set block to scroll 4 pixels

50 logic=5

60 scroll 1 : wait vbl

70 ink 0 : bar 32,0 to 35,100 : rem Get rid of picture trail

80 logic=physic

90 A$=screen$(5,0,0 to 320,100) : rem Capture pre-shifted block

100 rem merge it onto screen at right place

110 screen$(logic,0,0)=A$

120 A$="" : erase 5 : stop
```

So, what this routine is doing is putting our block on a sixteen-pixel boundary in a bank and scrolling it along from X Co-ordinate 32 to X Coordinate 36. Then we can merge it on screen, this takes it from its exact position and puts it on screen at its exact position.

So that's it, with a bit of work you can have a large block moving across the screen like a sprite. This is how to do it all if you don't have the missing link or extra extensions. In the missing link, you can set up your pre-shifted sprites using a special MAKE program and you can position blocks anywhere on the screen using the ppsc command from the Extra extension.

THE EXTRA EXTENTION TUTORIAL

As the name suggests – the Extra extension provides something a little extra. Improved commands and a few you may find useful. Without further ado let's get started.

EXTRA

Typing this command displays a list of commands the extra extension provides. Useful if you want a quick way of checking how a certain command works.

LEXTRA

Same command but prints the command list to the printer rather than the screen.

PRNTR

Check to see if your printer is ready to print. Returns a value of -1 when ready and 0 when it isn't.

```
10 repeat
20 home : print "Waiting for Printer"
30 until prntr=-1
40 print "Printer is ready"
```

SCREEN DUMP

It does just what it says – it dumps (prints) a picture on-screen onto your printer, the same as the STOS hardcopy command.

```
10 mode 0 : key off : hide : curs off : flash off
20 if prntr=0 then print"Press the online switch."
30 if prntr=-1 then screen dump
40 goto 20
```

DEPACK TINY

This command unpacks pictures in TINY format (TN1). Been known to clash with the Depack command from the Missing Link extension. Disable Missing Link if you want to use this command.

To unpack a TINY picture, you must first reserve a bank to the size of the TINY picture then BLOAD it in.

```
reserve as work 5,9000 : bload"PIC.TN1,5
```

You can then choose to unpack it to the screen or a screen bank.

```
depack tiny start(5),physic or depack tiny start(5),start(6)
```

CONVERT IFF

This command simply converts an IFF picture file to a PI1 or NEO picture file. The IFF file must be unpacked. The command is…

convert iff start(5),physic,palst

This line converts the IFF picture in bank five and unpacks it to the screen which is used as workspace, you can also use a screen bank instead of a screen. The PALST value is used to decide whether you want to get the palette of the picture. Values are 0 for "don't get palette" or 1 for "get palette".

BLUR

When broadcasters don't want you to see something on television they block it out using a jumbled pixel effect. The blur command allows you to do the same. The command looks like this.

blur ADDRESS,XPOS,YPOS,XBLOCK,YBLOCK,XSIZE,YSIZE,TYPE

ADDRESS: Can be a screen (IE: physic) or a memory bank.

XPOS AND YPOS: The X and Y coordinates of the block of the picture to be blurred.

XBLOCK AND YBLOCK: The size of the block to be blurred.

XSIZE AND YSIZE: The size of each zoomed pixel that makes up the block.

TYPE: Tells the command which pixel to use to get the colour of the resulting block – ranges from 1 to 5.

DESHADE

deshade ADDRESS,IGNORE,COLOUR

The DESHADE command changes all the colours at ADDRESS which can be a screen, or a bank, to COLOUR. The IGNORE value allows you to specify which colour in your palette is not to be changed.

10 key off: mode 0

20 load"CAR.PI1?

30 deshade physic,0,6

HREV

hrev S$

HREV allows you to flip a picture in a string horizontally. The picture block must first be put in a string (S$) with SCREEN$.

10 key off: curs off : mode 0

20 S$=screen$(5,32,32 to 112,100)

30 hrev S$

40 screen$(physic,0,0)=S$

VREF

vrev S$

As HREV but flips the screen vertically.

PPSC (pixel perfect screen copy)

This command works like Screen copy only that it allows you to copy pictures anywhere on the screen, it is not limited by the ST's sixteen-pixel boundary problem. Can become quite slow when copying large areas of the screen.

ppsc Source,Des,X1,Y1,X2,Y2,X,Y,Plane

The Plane parameter allows you to choose how many screen planes to draw the picture in. It's usually set to four for the full set. The Source can be a screen or a bank where you copy from and Des is the destination of the screen – again a screen or a bank.

FMT DISC

fmt disk drive,size,buffer

This command allows you to format a disk to different sectors. DRIVE is 0 for drive A or 1 for Drive B. SIDES can be 0 (single-sided) or 1 (double-sided). BUFFER is a memory bank reserved to at least 5000 bytes for the command to use as its workspace.

SET SCREEN

set screen HZ Hertz

This command works like the Frequency command in STOS only this command will work fine in a compiled program. Hertz can be 50 or 60.

OS VERSION$

Every time a new ST comes out, its TOS is updated. OS VERSION$ is a variable that finds out what the TOS version of your ST is. You use it by letting a variable equal it using the VAL command to find this out. For example, let's use a line that checks if your ST has TOS 1.62.

10 OS#=val(os version$)

20 if OS#=1.62 then print "You have an STE with TOS 1.62?

MEM CONFIG

This command is used to tell you how much memory your ST has in Kilobytes. Usually, the ST measures memory in bytes rather than K so a one meg ST will have around 800000 or something like that. It works by taking the number of bytes of memory your machine has and dividing it by 1024, which is the size of a Kilobyte in bytes. Try this routine.

10 D=free/1024

20 print D

This routine tells you how much memory you have in K's (Kilobytes). However, the "free" command takes away the memory that's been used by STOS and anything loaded in. But "mem config" tells you how much memory in K you have altogether without anything taking it up. It goes like this.

10 D=mem config

20 print "The total K of memory on this ST is ";D

CARTRIDGE INPUT

If you have a sampler cartridge then you can create some nice disco effects with this command. It's a variable that returns the value from the cartridge between 0 and 255. You could use it to create a simple VU meter like this.

10 key off : hide on : flash off : mode 0 : curs off

20 logic=back

30 X=cartridge input-127

40 ink rnd(14)+1

50 cls logic

60 bar 0,0 to X,20

70 screen swap : wait vbl

80 goto 30

So, in this case, the higher the value the longer the bar will go. Note you have to always subtract 127 so that nought becomes stable.

POWER

power (A,B)

The POWER command is a new faster and more trusted version of the ^ symbol. The idea is to multiply a number so many times against itself. Let's take this example.

3^5

33333

Each sum gives us the answer 243. What it does is take a number and raise it to the power of another number. So, in the last example we are multiplying three to the power of five (3^5) which is the same as the second example but shorter reading and faster. So, to do this with the POWER command we just go.

A=POWER(3,5)

Which is the same as…

3^5

The next command is…

DISC SIZE

disc size (DRIVE)

This command is similar to the DFREE command only it returns the total size of the disk. If you had a file on disk which was 49000 bytes long then DFREE would return the space left on the disk minus 49000 bytes. Where DISC SIZE returns the size of the disk itself in bytes rather than report the free space left on the disk.

10 D=disc free(0)

20 if D=802300 then print ”This disk is 802300 bytes in length”

It’s useful if you wanted to check if an inserted disk is a nine-sector disk by getting the size of the disk in bytes and if DISC FREE returns that number then it’s a nine-sector disk. The DRIVE parameter has two values. That’s 0 to check the disk in drive A and 1 to check the disk in drive B.

DISC FREE

disc free (DRIVE)

This is just the same as DFREE only it allows you to check a certain drive.

10 D=disc free(1)

20 print D;”bytes free on disk B”

EXTRA is not the best extension out there but it does have some commands you will find useful.

FIRST STOS GAME FOR NEWBIES

Well, you've opened the STOS manual. You've read it and seen all the clever commands and accessories you can use. You've read and heard that all this can enable you to write your very own masterpiece. But how do you start, what do you do, how do you tell STOS to do this? These are the questions the beginner asks. In this article, I will show you how to get started and even write a simple game.

How do we start? Well, first we need to learn some commands. A program is just a group of commands that tell STOS what to do. We start with simple commands and we type out the examples in the STOS manual. This helps to give us a good idea of commanding STOS and telling it what to do.

Let's say we wanted STOS to print something on the screen over and over again. We can use the PRINT and GOTO commands like this…

```
10 print ”STOSSER”

20 goto 10
```

Here we have a simple program. First, it prints the word STOSSER onto the screen, and then it goes onto the next line that tells STOS to go back to line 10 and print STOSSER on the screen again.

So, as you can see, we have grouped two commands to make a small program a small program like this is better known as a routine. A program such as a game is just a large group of routines. Let's try a routine that counts up to ten.

```
10 for X=1 to 10

20 home : print X : wait 5

30 next X
```

What we have here is a loop that adds one to the variable X, prints it on the screen, then stops when X reaches 10. The HOME command is used to tell STOS to keep the count printing on the same part of the screen, and the WAIT command stops STOS from printing too fast.

So once we have learned the commands we can easily group them. Once you've been learning them, typing out examples, and looking at other routines, you'll find it all comes clearer.

Let's program a small game. First, we have to decide what the program is going to be, and what it does. The game in question is a Guess the Number game where STOS will first choose a random number then ask the player to guess it. If the player's guess is wrong then STOS will tell him so. First…let's set up the screen.

```
10 key off : hide : mode 0

Rem Now to start printing the information we need on screen.

20 locate 0,1 : centre”NUMBER GUESSER”
```

Rem Now we need to tell STOS to choose a random number, like this.

30 RN=rnd(10) : if RN=0 then goto 30

This line will choose a different number from 1 to 10 every time STOS comes across it. With programming we are sometimes faced with problems, in this case, the RND command will sometimes choose nought as a random number. The problem is that we only want numbers from 1 to 10. So, therefore, we add an IF THEN statement to tell STOS that if it chooses nought…then go back and try again.

When STOS has chosen a number between 1 and 10, it needs to ask the player to guess the number.

40 locate 2,4 : print”I am thinking of a number between 1 and 10?

50 locate 2,5 : print”Can you guess what it is?”

Next we need a way of letting the player enter his guess.

60 input A

This command waits for the player to enter a number and press return. Now so far, we have the RN variable which holds the random number chosen by the game, and the A variable which holds the number chosen by the player. But what if the player has typed in a number which is out of range? We can check for this with these two lines.

70 if A<1 then print”Your guess is too low, try again” : goto 60 80 if A>10 then print”Your guess is too high, try again” : goto 60

So, if the number in the A variable is out of range, then the game informs the player to try again then goes back to the input at line 60. Now we need a line to check if the player has guessed the right number.

90 if A<>RN then print “That’s not it, try again” : goto 60

This line checks if the A variable contains the same number as the variable chosen by the game. Here we are telling the game that if the player's guess is other than the number the game has chosen then to give them a chance to try again.

Finally, we need to check if the player has guessed the right number.

100 if A=RN then print “Well done, you’ve guessed the number” : stop

So there we have it. A small game produced by just a group of routines. We can easily improve it, such as getting the game to loop back to the start after the player wins. We could also add extras such as a score by simply adding points to a variable and printing it on the screen like this.

110 SC=SC+10 : print “Your score is now:”;SC

120 wait 50 : cls : goto 20

I recommend you download The Beginner Guide to STOS Basic by MT Software as this is much more detailed.

HOW TO CREATE YOUR OWN DISK PROTECTION

Protecting a disk from being copied probably involves altering the way the ST reads a disk. A protection is a piece of machine code which is stored in the boot sector of a disk and if it’s read by something other than a ST it can cause all things to happen like damaging the copy on the copy disk. Whilst programming the boot sector takes machine code programming there is a simply way we can use STOS to do it.

Note this protection can only be used on an unformatted disk and will not fool the more advanced copiers like Procopy or Blitz Turbo. However, to someone who only uses basic copiers like Fastcopy 3 or copies the files across in Gem then the protection can stop them.

Basically, what we are doing is putting a magic number onto an empty track on a disk. Your program then reads the number and if it’s not there then the disk is a copy and you can then tell your program to reset or do something nasty like format the disk. You may think that something like Fastcopy will copy all the tracks including the one with the number on. Well, this is what we use an unformatted disk for. Fastcopy will not read an unformatted track.

This is how to set up a disk for the copy protection. First load up Fastcopy 3 or Fastcopy Pro. In the right hand corner, you will notice that the program allows you to set the start and end tracks to format the disk. Set the end track to 72 and format your disk with Fastcopy. When you scan the disk you'll notice that Fastcopy thinks the disk has only 72 tracks and will only read that. In fact the bootsector will tell your ST the same thing.

Now set the start track to 79 and click on format again. This will format track 79 which is the track our magic number will appear on. You now have a disk with tracks 0 to 72 formatted, tracks 73 to 78 unformatted, and track 79 is formatted. Now click on the scan option again and you'll see the scan option only scans the first 72 tracks. If you set Fastcopy to read all sectors then it will refuse to read the next few tracks after track 72. And after clicking on retry and finding about two bad tracks the user will probably just think the end of the disk is damaged. Therefore, when the copy disk is made it will just be a normal disk with all tracks formatted. All the files from your disk will be on - but not the magic number.

Now to put the number on the track we use the FLOPWRT command which will write a number onto track 79. This command is from the MISTY extension. Insert your protected disk and run this program.

```
10 reserve as work 10,512

20 poke start(10),100

30 flopwrt start(10),10,0,79,1,0
```

This will write 100 on track 79 of a ten-sector disk in drive A.

Copy all your files by hand onto the disk. Using a copier will obviously wipe out the magic number. Next enter this routine at the start of your program.

```
10 reserve as work 10,512

20 floprd start(10),10,0,79,1,0

30 PROTECT=peek(start(10))

40 if PROTECT<>100 then print "You have copied this disk" : wait 400 : stop
```

A really nasty thing to do is to run this routine when the game is in play. For example, when someones been playing the first level for about two minutes. After all, someone copying the game for his mate will just check that the game loads before sending it to him. Imagine getting a really good score when the game stops and calls you a pirate. You can use this routine for this effect.

```
200 if SCORE>10000 then goto 1000 : rem the disk checking routine.
```

As I said this method will not work with more powerful copiers that read all the tracks, skipping the unformatted ones – but will work with the less powerful ones.

HOW TO PROGRAM AN UNLIMITED INPUT ROUTINE

Ever found that you are doing some routine that requires the user to enter some kind of string of characters and the input command doesn't seem to fulfil your requirements. Then why not program your own input routine. This is one of mine I have used in various routines such as my own STOS Adventure Creator. It works better because it allows you to move anyway in the text and change it. The STOS input command only allows a limited number of characters but this routine is unlimited.

```
10 key off : curs off : flash off : hide on : mode 1

15 windopen 1,0,0,80,25,1,4 : curs off

20 get palette (4) : wait vbl

25 dim PAGE$(1000)

30 fastcopy start(4),back : fastcopy back,logic 610 rem EDIT MODE

612 XW=8 : YW=1 : paper 1 : pen 0

613 locate XW,YW

614 while K$=""

615 K$=inkey$

616 wend

617 rem ~~~ CHECK CURSOR UP KEY

618 if asc(K$)=0 and scancode=72 and YW>1 then dec YW : locate XW,YW

619 rem ~~~ CHECK CURSOR DOWN KEY

620 if asc(K$)=0 and scancode=80 and YW<>21 then inc YW : locate XW,YW

621 rem ~~~ CHECK CURSOR LEFT KEY

622 if asc(K$)=0 and scancode=75 and XW>8 then dec XW : locate XW,YW

623 rem ~~~ CHECK CURSOR RIGHT KEY

624 if asc(K$)=0 and scancode=77 and XW<37 then inc XW : locate XW,YW

625 rem ~~~ CHECK RETURN KEY

626 if K$=chr$(13) then curs off : goto 636

627 rem ~~~ CHECK BACKSPACE KEY

628 if K$=chr$(8) and XW>8 then dec XW : locate XW,YW : print " "; : locate XW,YW

629 if K$=chr$(8) and XW=8 and YW>1 then dec YW : XW=35 : locate XW,YW : print
" "; : locate XW,YW 632 rem ~~~ CHECK FOR KEYBOARD INPUT

633 if XW=35 and YW=21 then stop

634 if asc(K$)>31 and XW<37 and YW<21 then print K$; : inc XW
```

```
635 if asc(K$)>31 and XW=36 and YW<21 then XW=8 : inc YW
636 K$="" : goto 613
```

HOW TO STORE VARIABLE INFORMATION ON A DISK

When I write a program, I sometimes find it easier to store variables on disk rather than inside the program. Using a few variables, it's okay to store them in a program but what about storing lots of information in variables. This makes your program a lot longer. Let's say, for example, I was writing an adventure game and I wanted 100 location descriptions. In my game, I could have a routine like this.

```
10 dim LOCATION$(100) 20 for X=1 to 100 : read LOCATION$(x) : next X

1000 rem LOCATION DESCRIPTIONS

1010 data "This is location one."

1020 data "This is location two."
```

And so on, adding a hundred lines to STOS. How about doing it this way?

```
10 LOCATION=1

20 print "Type in the description for location "+str$(LOCATION)

30 input L$

40 F$=file select$("*.DAT")

50 open out #1,F$

60 print #1,L$

70 close #1

80 cls : inc LOCATION : goto 20
```

Doing things this way with this separate routine allows me to enter the description as I would see it on screen, all nicely spaced out, rather than playing about spacing out the words in a data statement. I could save each file out under names like LOC1.DAT for location one and LOC2.DAT for location two etc. Once all descriptions are saved then I could put a routine in the game to load each file when I need it. For example.

```
10 LOCATION=1

15 F$="LOC"+str$(LOCATION)-" "+".DAT"

20 open in #1,F$

30 input #1,L$

40 close #1

50 print L$ : rem print Location

60 print : inc LOCATION : goto 15
```

Here's another tip for a program that needs loads of zones on-screen on loads of pictures. Just simply type out a program that allows you to create the zone data using just the mouse.

```
10 key off : flash off : curs off : mode 0
15 dim X1(10,10),Y1(10,10),X2(10,10),Y2(10,10)
20 ZP=1 : ZA=1 : rem ZP=Zone Picture ZA=Zone Area
30 F$="PIC"+str$(ZP)-" "+".PI1?
40 load F$ : screen copy physic to back
50 repeat : until mouse key=1 : X1=xmouse : Y1=ymouse
60 wait 40
70 repeat : until mouse key=1 : X2=xmouse : Y2=ymouse
80 rem X1,Y1 : Top left hand co-ordinates of zone
90 rem X2,Y1 : Bottom right hand co-ordinates of zone
100 X1(ZP,ZA)=X1 : Y1(ZP,ZA)=Y1 : X2(ZP,ZA)=X2 : Y2(ZP,ZA)=Y2
110 if ZA<10 then inc ZA : goto 50
120 if ZA=10 and ZP<10 then inc ZP : ZA=1 : goto 50
130 open out #1,"ZONE.DAT"
140 for X=1 to 10 : for Y=1 to 10
150 print #1,X1(X,Y),Y1(X,Y),X2(X,Y),Y2(X,Y)
160 close #1
```

Once this is done you simply load this file into your program using 'Open In' and you can set your zones easier. For example.

```
10 key off : curs off : flash off : mode 0
20 dim X1(10,10),Y1(10,10),X2(10,10),Y2(10,10)
30 open in #1,"ZONE.DAT"
40 for X=1 to 10 : for Y=1 to 10
50 input #1,X1(X,Y),Y1(X,Y),X2(X,Y),Y2(X,Y)
60 close #1
70 ZP=1
80 unpack 5 : rem unpack picture one
90 for ZA=1 to 10
100 set zone ZA,X1(ZP,ZA),Y1(ZP,ZA) TO X2(ZP,ZA),Y2(ZP,ZA)
110 next ZA
120 repeat : CH=zone(0) : until mouse key=1
```

There are loads of things you can do to improve the length of your programs and makes things easier. One thing you can do is write a routine to record the X and Y coordinates of alien movement instead of typing in the coordinates in data statements by hand.

THE MISSING LINK MAKE PROGRAM

The missing link doc file talks about different kinds of banks for things such as bobs, joeys, blocks, etc. But how do we make those banks in the first place? The answer is simple. On the missing link disk is a file called MAKE.BAS which allows you to convert sprites to other formats.

The missing link uses a new type of sprites called bobs. These are called pre-shifted sprites. The normal STOS sprites work like this. To get past the problem of the ST's sixteen-pixel boundary the sprite has to be pre-shifted. This means so many copies of the sprite have to be made and each one is scrolled by one pixel to the right until there are sixteen copies of one sprite image made before it goes on screen.

This all has to be done before the sprite can be moved or placed on the screen to position it anywhere correctly. This takes a fair bit of processor time which explains why sprites are often jerky and slow. So the routine goes and makes sixteen copies, place the sprite on the screen, make another sixteen copies, place the sprite on screen at the next pixel, and so on. But there is a better faster way of doing this.

The difference between STOS sprites and pre-shifted sprites is that pre-shifted sprites are pre-shifted once and held in memory whilst STOS sprites are pre-shifted as they move across the screen which means they take up less memory than pre-shifted sprites.

Even though STOS sprites make sixteen images of themselves we can decide how many images of pre-shifted sprites are made. We can make either 16, 8, 4, 2, or 1 image. Might you the fewer images you use the more they jerk when moved. This can be fixed by moving them at different steps. So, we need to calculate how many pixels to move an image. So, if we wanted our sprite to move in steps of one pixel we would need to make sixteen images, one for each of the sixteen pixels of the boundary. This is worked out like this.

16/1=16

We want to move the sprite at one pixel at a time so we divide it by sixteen and we get the answer sixteen. This means we have to make sixteen images of the sprite for it to be moved by one pixel at a time. Now let's suppose we wanted to move it by two pixels.

16/2=8

This has now halved the size of the image because we're now moving the image at two pixels at a time so only half of the images have to be made. Look at these other examples.

16/4=4 : Move sprite at 4 pixels so make 4 images of each sprite.

16/8=2 : Move sprite at 8 pixels so make 2 images of each sprite.

So, all we are doing is simply taking the number of pixels we want the sprite to scroll and dividing it by sixteen to found out how many images we need to make. Note that if your sprite is to move up and down the screen only then we only need to make one image as the sixteen-pixel boundary only applies to moving across the screen.

But how do we make these images in the first place? Well, that's where the MAKE program comes in. First, load it up and you are presented with menus covering different convert options. The one we are interested in at the moment is the one to convert our STOS sprites to pre-shifted sprites (bobs). From the LOAD menu click on SPRITES and load your sprites. Next, go to the MAKE menu and click on MAKE BOBS. Here we see three options.

MAKE BOBS

IMAGES

QUIT

Before we convert the sprites to bobs, we need to choose the number of images we need of each one. Position the mouse pointer over the word IMAGES and press the left mouse button. You will now see the word IMAGES with a number next to it and the word SPRITE above that with its image number next to it. Move the mouse pointer over either word and click on either mouse key. Notice how the left and right mouse buttons step through each number of each option.

Select sprite one and image four by clicking on SPRITE till its number says 1 and on IMAGES till it says 4. This means we have set MAKE to make four images of sprite one. If we wanted to make four images of all the sprites then we can simply click on IMAGES with both mouse buttons. Note that if we now step through the sprites, we see they are all set to be converted to four images each.

Now all the images are set we can proceed to make them into bobs. Click on EXIT from the IMAGE selection screen and click on MAKE BOBS to start the pre-shifting. Notice how MAKE shows the sprites pre-shifting on-screen while it's doing it. How long it takes depends on how many images are being pre-shifted.

When the program stops, click on EXIT to go back to the main menu. Our sprites are now converted to bobs and all we have to do now is save them. From the SAVE menu click on BOBS and save them under their new name making sure the extension is .MBK. It's important to remember that the more images made of each sprite the bigger the bank is so make sure there's enough room on the disk. The MAKE program has just turned the sprite bank into a bob bank for use with the bob commands. We can now use these bobs in a routine.

```
10 key off : flash off : hide on : curs off : mode 0

20 rem First load the bob bank into memory bank 5

30 load "bobs.mbk",5

40 rem Place bob one on screen and move it along by 4 pixels

50 logic=back : XBOB=10 : YBOB=100

60 repeat

70 wash logic,XBOB-10,YBOB-10,XBOB,YBOB
```

```
80 bob logic,start(5),0,XBOB,YBOB,0

90 screen swap : wait vbl

100 XBOB=XBOB+4 : until XBOB>300
```

This routine loads our BOB bank into bank five and the rest of the routine zooms it across the screen. See how fast and smooth it moves? Notice also that unlike sprites the bobs can be placed into any bank and they can either be loaded in every time the routine is run or it can be left in there without loading all the time. The reason for WASH and SCREEN SWAP is because unlike sprites the bobs don't clear the trail behind them. Notice also that WASH is set to clear ten pixels behind and over the sprite to ensure no trails are left.

JOEYS

Joeys are only allowed one colour while bobs can use the full sixteen colours. Useful for single colour sprites such as bullets and there are faster as well. You can convert them in the same way as bobs.

WORLD BLOCKS

This option will convert our sprites to world blocks so they can be used by the world command. Same method as converting sprites to bobs.

LANDSCAPE BLOCKS

This option is the same as creating world blocks only that it converts sprites to landscape blocks for use with the landscape command. Note that there's no IMAGES option for this option, this is because landscape only moves its blocks up and down the screen so the sixteen-pixel boundary doesn't exist here so there's no need for pre-shifting. Just click on MAKE BLOCKS and save them.

TILES

Used by the tile and moziac commands. This option works the same as the bob option only that it creates tiles.

Note that these last four options all need to be saved with the MBK extension, like BOBS they are converted into special banks so they can be used by their commands. They are loaded in the same way as bobs.

PICTURE

This option makes up a picture of your sprites, to activate it you must first load the sprites then from the MAKE menu click on PICTURE, the program will then place your sprites on screen in order one after the other and make up a picture. You can then save the picture. The sprites must be 16×16 in size.

DIGIBANK

Normally you can play a sample using the digiplay command but should you be used to using STOS maestro then you may want to put a load of samples into one bank then this is where this option comes in. Let's look at each option, first from the MAKE menu

click on DIGIBANK and you'll be presented with a new options screen. Note that you can either select a number or click on the option.

LOAD SAMPLE

Allows you to load a sample into the bank.

SAVE SAMPLE

Unlike STOS maestro's accessory, this option allows you to save a sample out of the bank and on disk as a stand-alone sample.

LOAD DIGIBANK

When you've saved a bank of samples, it becomes a digibank so the digiplay command can understand it. This option allows you to load the previously saved bank for editing.

SAVE DIGIBANK

Saves out a digibank for use with the command.

CLEAR DIGIBANK

Clears the digibank in memory, all samples are lost.

PLAY SAMPLE

Choosing this option will take you to another screen, position the mouse over the number, and press either mouse key to step backward and forwards through the samples in memory. To play it press space and enter the speed. Press space to play it and the sample will play in a loop. Press space again for the main menu.

(UN)SIGN SAMPLE

This option will take you to the play screen. From here select the sample you wish to sign/unsign and press space for the main menu.

EXIT

Takes you back to the main menu.

PLAYING A DIGIBANK SAMPLE

Once you've created your digibank and saved it you can play it using the digiplay command like this.

digiplay 1,start(5),SAMPLE,10,1

The parameter SAMPLE is normally used for the length of a raw sample. In this case, it ranges from 0 to 49. If the size is less than fifty then the parameter SAMPLE is used for the number of the sample to play between 0 and 49. If SAMPLE equals more than 49 then the data in bank five is assumed to be a raw sample but if it's less than fifty then it's assumed to be a digibank. Here are two examples.

PLAY A RAW SAMPLE AT SPEED 10

digiplay 1,start(5),32000,10,1

PLAY A SAMPLE FROM A DIGIBANK AT SPEED 10

digiplay 1,start(5),4,10,

The Make program takes care of all your converting needs for the missing link extension. One thing I would recommend is that you pre-shift your sprites and blocks at four pixels as this is a good combination of speed and smooth movement.

THE MISSING LINK MAP EDITOR

The Missing Link Map Editor (EDDY) is used to set up the scrolling maps that the commands "world" and "landscape" uses. It works similar to the MAP accessory that comes with STOS only we have a larger area.

So, what does this map look like then? Well load one of the example files on the missing link disk and have a look. Try landscape.bas and run it. As you can see, pushing the joystick the screen scrolls up and down. This screen is made up of a set of blocks called landscape blocks which were made from sprites using the MAKE program.

Let's use EDDY to create our simple map. Load EDDY and run it. You will be presented with the editor screen; this is where our blocks are to be placed to build up a scrolling map as in the landscape.bas file. Press SPACE to go to the editor. From here we see various options which we'll look at soon. But first things first. We need to create some blocks to make the zone. So, exit from EDDY and load up the MAKE program, ie: load "make.bas". Run it and enter the STOS disk and select the load sprites option. Load the file MAP.MBK, this is a sprite bank containing sprites that could be used to make a game like gauntlet. Convert the sprites to world blocks making four images of each sprite. Once it's done, save the file as world blocks under the name of WBLOCK.MBK.

Reload EDDY, run it, and go to the editor. Click on the load world blocks option and load the newly converted file (WBLOCK.MBK). Once this is done the blocks will line up along the top of the screen. Click on one to select it, then press SPACE to continue.

Note it is a good idea to insert a blank space in the sprite bank before converting it. You can now click on this with the right mouse button to assign it to that button. So now if you put a block on the screen then you can erase it by placing the mouse pointer over that block and pressing the right mouse button.

Once you've clicked on a block to use and pressed SPACE you will be in the editor window. Place the mouse pointer somewhere and press the left mouse key. The block appears at that position. Click on it with the right mouse button if you've assigned a blank block to it and the block will vanish. Useful if you make a mistake.

Note that pressing UNDO will exit the program and pressing * on the number keypad will give you online help. I think this option works on the compiled version only though.

Making a scrolling map is like making a map with the map designer. Just click on the left mouse button to place the block and the right mouse button to erase it. Note that the first block in your sprite bank must be a blank space as EDDY uses this block to erase blocks from the screen, otherwise the map area is filled with these blocks. At the bottom of the screen are the coordinates of the mouse pointer. Try moving the pointer about and you'll notice that they go further than normal screen coordinates X 319 Y 199. So, as you can see, the map area is larger than the actual screen.

When making a landscape map then make sure the blocks at the end of the screen start at X co-ordinate 304, if you go beyond this then the blocks on the right-hand side will be cut off in the landscape routine. Note that due to a bug in this program you can't load landscape blocks into EDDY so to create a landscape map you must first make

your sprites into world blocks, make the map, make the sprites into landscape blocks and use them in your routine.

Note you can build up a series of blocks on screen then make a copy of them to paste elsewhere. Let's say you made a small maze and you wanted to place that elsewhere on the map then position the mouse at the top left-hand corner of the maze and press F1. Then position the mouse at the bottom right-hand corner of the maze and press F2.

You have just captured this maze. Note it works like the cut and paste feature in art packages. If you press F3 then that block will be erased from memory. If you wish to paste the maze block somewhere else on the map then position the mouse pointer where you want it and press F4. Note that when you cut the block a stream of colour covered it telling you how much of the area was captured.

This captured block is known as an area, and it can be saved for future use by pressing F9 to save it to disk. It can be loaded back later by pressing F10. Pressing F5 will fill the screen with copies of that area. Pressing BACKSPACE will fill the whole map with copies of the first block, useful for clearing the screen if the block is a blank space.

Once you have defined your map then you can see it by moving the mouse around. Once you are happy then press space to go back to the main menu and you can save it. There are two save options, one will save the map as world data and the other as landscape data. Note that you now have two files.

WBLOCKS.MBK……..the converted sprites we did before.

WDATA.MBK……..the data you saved as world data.

You can now put this map into a routine.

```
10 key off : hide :curs off : mode 0

20 load "WBLOCKS.MBK",5 : load "WDATA.MBK",6

30 a=palt(start(5))

40 X=0 : Y=0

50 repeat

60 if jleft then X=X-4

70 if jright then X=X+4

80 if jup then Y=Y-4

90 if jdown then Y=Y+4

100 world logic,start(5),start(6),X,Y,0

110 until X>1200 or X<16
```

Note there is a resize option that allows you to place bigger sprites on screen but you're best off keeping them 16×16 as they fit perfectly on screen.

Well, that’s it for this article. The best way to learn more about EDDY is to play around with it. Before I go, I’ll just say that the above scrolls a world map. Try converting your sprites to landscape blocks and loading them into the routine instead of the world blocks. Remember that these maps only scroll up and down.

THE MISSING LINK TUTORIAL

The Missing Link is, in my opinion, the best extension to be written for STOS. It covers a lot of areas of STOS programming including a set of extra commands plus better faster versions of old ones. In this tutorial, I shall take you step by step through the commands explaining in detail how to use them.

SPRITE COMMANDS

BOB X1,Y1,X2,Y2,0,1

BOB SCR,ADR,IMAGE,X,Y,0

JOEY X1,Y1,X2,Y2,0,0,1

JOEY SCR,ADR,IMAGE,COLOUR,X,Y,0

H=B HEIGHT (ADR,IMAGE)

W=W HEIGHT (ADR,IMAGE)

The BOB command is a new version of the SPRITE command. It is much faster and smoother, and there's no limit to how many you can have on the screen at the same time. This is a new method of sprite movement known as Pre-Shifting, and although it takes up more memory than normal sprites it is much better. The format of this command is:

bob SCR,ADR,IMAGE,X,Y,0

SCR is the screen to place the BOB on. Note that unlike SPRITE which is only displayed on the physic or logic screen, a BOB can be displayed on either the BACK screen, the PHYSIC screen, or even the LOGIC screen. It can also be placed in a memory bank.

ADR is the memory bank where the BOBs are held. Unlike sprites that can only be accessed from bank one, BOBs can be loaded into any bank. Note that we need to use the START command to tell STOS which bank the BOBs are in. So if the BOBs were loaded into bank 5, the variable ADR would be 'start(5)' and not '5' as we use with commands like 'screen copy'.

IMAGE is the number of the BOB to display on-screen which ranges from nought to the number of bobs in the bank. It's important to remember that when you convert your sprites to bobs, the image numbers are moved back by one place. So the first sprite would become BOB 0, the second would become BOB 1, and so on…….

X and Y are simply the X and Y coordinates of the BOB. Note that unlike sprites the hot spot for a bob is best positioned in the top left-hand corner. So, the BOB is placed on the coordinates of the hot spot. The last number (nought) doesn't do anything, it was there for future purposes but was never used.

bob X1,Y1,X2,Y2,0,1

This command is a new version of the LIMIT SPRITE command, only it limits a BOB to a certain part of the screen. To limit the bob means to set up an area on the screen

where the bob is to be visible. If it moves outside this area then it will vanish. Try this routine…

```
10 key off : hide : curs off : mode 0

20 load”bob.mbk”,5

25 A=palt(start(5))

30 box 50,50 to 150,150

40 bob 50,50,150,150,0,1

50 logic=back

60 XB=60 : YB=60

70 repeat

80 XB=XB+2

90 bob logic,start(5),0,XB,YB,0

100 wait 5

110 screen swap : wait vbl

120 until XB=170

130 goto 60
```

This routine will draw a box on the screen and move the BOB to the right by steps of two pixels. Note that when the bob moves out of the box it starts to vanish which could be used for great effects in games. As you can see, the coordinates of the box are the same as those of the limiting version of BOB meaning keep the bob between these coordinates.

The next command is the JOEY command. This command is similar to the BOB command only it’s used for sprites which are only one colour. For example a white bullet sprite. The format of the command is…

```
joey SCR,ADR,IMAGE,X,Y,COLOUR,0
```

The parameters of this command are the same as the BOB command, with the extra parameter called COLOUR. This is the number of the colour in the present palette that the joey is. For example, if the joey was white, and white was colour number ten in the palette then COLOUR would be ten. For unknown reasons, colour fifteen is the fastest.

```
joey X1,Y1,X2,Y2,0,0,1
```

This version of the command limits the joey to a certain part of the screen. It just the same as the BOB version only the two noughts are never used.

The last two commands are B WIDTH and B HEIGHT, the format is:

```
W=b width (ADR,IMAGE)
```

H=B height(ADR,IMAGE)

These commands return the size of a bob, in pixels. ADR is the number of the bank where the bob is stored and IMAGE is the image number of it.

```
10 key off : curs off : hide : mode 0

20 load”bob.mbk”,5

30 W=b width(start(5),0)

40 H=b height(start(5),0)

50 print”This bob is “;W;” pixels across”

60 print”This bob is “;H;” pixels down”

70 print”Total size is “;W;”X”;H
```

This command would be useful in finding out the size of a bob so we can calculate if it will fit on a certain part of the screen.

DEFINING BOBS AND JOEYS

If you look at the MAKE program, you’ll see that the options to convert bobs and joeys ask how many images it should make of each sprite. Well, the smaller the number, the less memory used, and the smaller the converted BOB bank. If you set the program to make eight images of a sprite, then they will move quite smoothly if you move them at two pixels at a time. Lower numbers mean you would have to move them in bigger steps for them to move smoothly. The same applies to JOEY’s, although these would normally be used as bullets which need to be fast so you can make about two images of each one.

MAPPING COMMANDS

WORLD X1,Y1,X2,Y2,0,1

WORLD SCR,BLOCKS,MADR,X,Y,0

LANDSCAPE X1,Y1,X2,Y2,0,1

LANDSCAPE SCR,BLOCKS,MADR,X,Y,0

SETBLOCK MADR,X,Y,BLOCK

R=WHICH BLOCK(MADR,X,Y)

REPLACE BLOCKS MADR,BLOCK1,BLOCK2

R=BLOCK AMOUNT(MADR,BLOCK)

XY BLOCK MADR,XADR,YADR,BLOCK,NUM

X=X LIMIT(MADR,X1,X2)

Y=Y LIMIT(MADR,Y1,Y2)

A=MAP TOGGLE(MADR)

These ‘mapping’ commands are used for scrolling the screen. Note that you could use ‘def scroll’ for this purpose but these new commands make scrolling a lot easier. For a start: scrolling left and right can be done in steps of less than 16 pixels thus creating smooth scrolling. It also solves the nightmare of moving sprites across a scrolling background. Let’s look at the first command.

WORLD,X1,Y1,X2,Y2,0,1

The WORLD commands allow you to scroll the screen left, right, up, and down. This version of the command allows you to specify how much of the screen you wish to use as the scrolling area. This is useful for having the top half of the screen as the scrolling area and the bottom half as the game’s scoreboard. If you’ve seen how the STOS ‘limit sprite’ command works you’ll see that the mouse is limited to that part of the screen, it’s like having it trapped in an invisible box.

This is how this version of ‘world’ works. It traps the scrolling map in an invisible box. X1 and Y1 hold the start co=ordinates of the screen area to be trapped. In other words, the top left-hand part of the screen. X2 and Y2 hold the end coordinates of the box – the bottom right-hand part of the screen. The last two parameters have no function at all.

To create these scrolling areas known as maps we first have to define one. We can do this using the EDDY program supplied with the missing link package. This is similar to using the ‘Map Definer’ supplied with STOS only we don’t use normal STOS sprites to create the map. We use the MAKE program to make the sprites into world blocks so they can be loaded into EDDY. Note that even though the EDDY manual tells us that we can load ‘world’ and ‘landscape’ blocks into the EDDY definer…we can only load ‘world’ blocks into it.

WORLD SCR,BLOCKS,MADR,X,Y,0

This is the command that scrolls the map we defined with EDDY. Note that we need two sets of data installed before we can use this command. These are World Blocks and Map Data.

SCR: This is the screen to display the map on, it can be either LOGIC, PHYSIC, or BACK.

BLOCKS: This tells the world command where our world blocks are, loads them into a memory bank, and uses start(bank-number).

MADR: The bank number of the map data saved from EDDY. It must have been saved as ‘world data’. Again use start(bank-number).

X and Y: The X and Y coordinates of the map's starting point. The nought at the end serves no purpose.

This routine shows the use of the two commands.

```
10 key off : hide : flash off : mode 0

20 load”BLOCK.MBK”,5 : rem load world blocks into bank 5

30 load”MAP.MBK”,6 : rem load world map data into bank 6
```

```
40 logic=back : X=0 : Y=0

45 world 32,10,288,190,0,1

50 repeat

60 world logic,start(5),start(6),X,Y,0

70 if jleft and X>0 then dec X

80 if jright and X<1500 then inc X

90 if jup and Y>0 then dec Y

100 if jdown and Y<2000 then inc Y

110 screen swap : wait vbl

120 until fire
```

If we look at lines 80 and 100 we see that the program is checking if the X and Y variables are less than a number that's higher than the coordinates of the actual screen. This is because the variables are the coordinates of the scrolling area of the map and not the screen.

Note that the first version of the WORLD command must have its X coordinates in steps of 16 pixels due to a bug in the ST's registers.

LANDSCAPE

The LANDSCAPE command is the same as the WORLD command except that it can only scroll its map in two directions. It's only used for games that only scroll up and down. The parameters are the same as those of WORLD, but the X coordinates cannot be changed in a loop, only the Y coordinates. X is just used to set the X starting point of the map to be scrolled.

Note: as with the first version of WORLD, the first version of LANDSCAPE must have its X coordinates in steps of 16 pixels. Note that we can use the last routine to see these commands in action. Just convert your sprites to LANDSCAPE blocks, load them into EDDY and make your map making sure you don't go over the X coordinates of 304, and make your map downwards, not exceeding the 320 X co-ordinates. Save your map data as landscape data, load your landscape blocks into bank five, and new landscape map data into bank six. Remove lines 80 and 90 and run the program. As we can see, we can only move up and down.

Due to the bug in EDDY, we have to make our sprites first into world blocks, then landscape blocks. Load the world blocks into EDDY to make the landscape map, then load the landscape blocks into our example routine. Bit of a pain but there we go.

R=WHICH BLOCK(MADR,X,Y)

As you may have noticed, we have been using these things called blocks (converted sprites), to make the world and landscape maps. So we can now use this command as a form of collision detection. The MADR variable holds the bank number of the map

data used by the world and landscape commands, and X and Y are the coordinates of the block we're checking for. R holds the block number. So really it's like checking for a sprite entering a zone without having to set the zone first. If we look at the screen in EDDY where we select our blocks, we can count along to a certain block to find its number in the row. Note that the row starts from nought to the number of blocks in the row, just like the BOB command. So if we defined, say a diamond as sprite number one, this would be block number 0.

So what we want to do is place our diamond blocks in certain places of the map as we define it, then tell STOS that when our BOB touches it, to detect a collision. Add this line to our example.

105 BL=which block (start(6),XBOB+14,YBOB+12)

The variables XBOB and YBOB hold the X and Y coordinates of the bob as the bob touches a certain block – the variable BL would contain the row number of it. So, if the bob touched the diamond, then BL would be set to nought. If the bob touched the next block in the row then BL would be set to one. Let's say we defined two sprites as part of the map and converted them to map blocks. We could use this routine to check which block has been touched. Try these lines…

106 if BL=0 then print "You have found the diamond."

107 if BL=1 then print "You have hit the wall."

Where the wall is the second block in the world/landscape blocks.

SET BLOCK MADR,X,Y,BLOCK

Each block in the map is set to either one if it exists, or nought if it doesn't. This means that if a certain block was in the map at coordinates X and Y, it would be set to one, or nought if it wasn't there. We can use this command with WHICH BLOCK to find a block. When it's found we can set the block to nought which will then erase the found block off the screen. MADR is the bank number of the map data, X and Y are the coordinates to check at and BLOCK is the row number of the block to remove from the screen. Example…..

10 BL=which block(start(6),XBOB,YBOB)

20 if BL=1 then set block start(6),XBOB,YBOB,0 : bell

REPLACE BLOCKS MADR,BLOCK1,BLOCK2

This command does as it says….it replaces one type of block with another. This is useful if say: you wanted to change all the diamonds on the screen to money bags for example. We can have our diamond block as block one in the row and the money block as block two, then change the diamonds into money bags like so….

100 replace blocks start(6),1,2

R=BLOCK AMOUNT(MADR,BLOCK)

This command is useful for checking how many times a certain block appears on a map. If we had our diamond and money bag blocks still in the same place and we put 20 copies of our diamond on the map screen, then we could use this command to find how many diamonds are on the screen. Try this example…..

100 DIAMONDS=block amount(start(6),1)

110 if DIAMONDS=0 then print ”All diamonds are collected”

So BLOCK holds the number of block ones (diamonds) there are on the map. As each one is removed with SET BLOCK, the DIAMONDS variable will decrease by one, and line 110 checks if all diamonds have gone.

XY BLOCK MADR,XADR,YADR,BLOCK,NUMBER

If we wanted to store all the X and Y coordinates of a certain block in an array this command will do it. So if we wanted to set the X and Y coordinates of the diamond…(block 1), we would use this little routine….

10 A=block amount(start(6),1)

20 dim XBL(A),YBL(A)

30 xy block start(6),varptr(XBL(0)),varptr(YBL(0)),1,A

40 for X=0 to A : print XBL(X),YBL(X) : next X

XADR and YADR are the X and Y arrays to put the coordinates in, BLOCK is the row number of the block we want to get the coordinates of, and NUMBER is the total number of block ones found in the map. This can be used if you had several diamonds and you wanted each one to score different points. Using the arrays you can see which diamond block has been collected and update the score with the certain diamonds points….like so

50 if A=1 and XBOB=XBL(1) then SC=SC+40

60 if A=1 and YBOB=YBL(4) then SC=SC+100

The next commands are:

X=X LIMIT(MADR,X1,X2)

Y=Y LIMIT(MADR,Y1,Y2)

These commands inform STOS how large in width and height the map is. X1 is the X start of the map, X2 is the X end of the map, Y1 is the Y start of the map, and Y2 is the Y end of the map. X and Y hold the end coordinates of the map. MADR is the bank number of the map data. This can be used to check if a bob is still within the X and Y areas.

M=MAP TOGGLE(MADR)

A nice simple one here. If the data in MADR is world data, then this command will convert it to landscape data, or vise versa…

```
10 load ”map.mbk”,6 : rem World data

20 N=map toggle(start(6))

30 print ”The world data is now landscape data”

40 N=map toggle(start(6))

50 print ”The landscape data is now back to world data”
```

This is useful if you had some levels in your game that use the landscape command, and some that used the world command.

TEXT COMMANDS

text SCR,FONT,TEXTADR,X,Y

TEXTADR=string(NUM)

The TEXT command is quite simply a replacement PRINT command. It has several advantages over PRINT. It only prints text on one bitplane which makes it faster than PRINT. It can be printed on any screen or memory bank. It can use other fonts without using a window and you don’t need to use the LOCATE command as it has one built-in.

The format of the command is……

text SCR,FONT,TEXTADR,X,Y

SCR: The address or bank number to display the text.

FONT: The FONT to print in……this can be the normal STOS default character set fonts or a font “bloaded” into a bank. Values 0 to 2 are the default LOW/MED/HIGH RES fonts in STOS, and 3 onwards is a bank.

TEXTADR: Unlike PRINT, we can’t just print a normal string or number, we need to put it in a variable first. So TEXTADR is the variable name that contains the string or number.

X & Y: The X and Y position of the text in text coordinates.

Let’s have a look at what it can do. We’ll use it to print a message on the screen, so we put the message in a variable.

```
10 T$=”STOSSER…..The only monthly STOS diskzine.”+chr$(0)
```

Now we use TEXT to print it on the screen.

```
20 text logic,0,varptr(T$),2,10
```

Note the use of the VARPTR command? This is because the TEXT command needs to find out the address of T$ before it can print it. Note that to use a string variable with TEXT we must add a CHR$(0) at the end….but not with number variables.

Run this program, and the message in T$ will be printed on the LOGIC screen with the LOW RES character set (FONT) at coordinates 2,10. Now let's print in a newly defined font.

First, we install our extra character set into bank five in the normal way by either using the QUIT AND GRAB option from the Font Definer or using this line.

reserve as set 5,2322 : bload”newset.mbk”,5

Now simply replace line 20 to……

20 text physic,3,varptr(T$),2,10

Remember that font 2 is the HIGH RES default character set in STOS so to use one from a bank we change it to 3. Note that we’ve also changed the command to print T$ on the PHYSIC screen. We can easily change the command to print T$ on a screen stored in a memory bank by simply reserving a bank as a screen in the normal way then changing SCR to start(bank number).

15 reserve as screen 5

20 text start(5),0,varptr(T$),2,10

Although the above example will print a string of text we can’t directly use it to print a number. So what we need to do is define a number variable then convert it using the STRING command.

TADR=string(number variable)

This command is a new faster version of the STR command. As TEXT only prints a string variable we need to convert the number variable to a string variable. Look at the following program.

10 T=12345

20 TADR=string(T)

30 text logic,0,TADR,2,10

Line 10 puts the number 12345 into the ‘T’ number variable. Line 20 puts it in the TADR variable as a string variable. And line 30 prints it onto the logic screen. Note the absence of VARPTR. We don’t need it to print a number as the STRING command has already found the address of the newly converted string.

The TEXT command has a large bug. You can only print in one pen colour: therefore using the PEN command does not affect the printed string. Unless Top Notch got round to fixing it since I wrote this tutorial.

MISCELLANEOUS GFX COMMANDS

wipe SCREEN

tile SCR,TADR,IMAGE,X,Y

mozaic SCR,TADR,IMAGE,X1,Y1,X2,Y2,X,Y

spot SCR,X,Y,COLOUR

reflect SCR1,Y1,Y2,SCR2,Y3

wash,SCR,X1,Y1,X2,Y2

blit SCR1,X1,Y1,X2,Y2,SCR2,X1,Y1

m blit SCR1,X1,Y1,X2,Y2,SCR2,X1,Y1

First command of this section is…

wipe scr

A nice easy one to start with. The WIPE command is a new version of CLS. Its advantage is speed, it's much faster than CLS, in fact about twice as fast. The variable "scr" can be the back, physic, or logic screen. It can even be a memory bank. The command works in the same way as CLS with the ability to clear any screen or bank.

tile scr,tadr,image,x,y

This one scrolls a wall of sprites in different directions on the screen. It was used in earlier STOSSER shells. It's also used by various demo coders. The command draws a wall of tiles that are converted sprites onto screen SCR. The TADR parameter is the memory bank where the tiles are held. IMAGE is the sprite number you wish to build into a wall and x and y are the screen coordinates to start the wall from. Note that like bobs the images in a tile bank start from nought instead of one. So if you wanted to use sprite one in your sprite bank then you would pass image as nought. When the sprites are converted to tiles using the MAKE program then all the images are moved back one place, so sprite one becomes tile nought, sprite two becomes tile one, and so on. Note also that the sprites must be 16×16 in size and this version of the command fills the whole screen with the wall of tiles.

```
10 key off : hide on : curs off: mode 0

20 for X=1 to 300

30 tile logic,start(5),0,X,0

40 next x
```

The next command is:

MOZIAC scr,tadr,image,x1,y1,x2,y2,x,y

This is another version of the TILE command. Only this one allows you to limit the wall to part of the screen instead of using all of it. The parameters are the same as TILE but with the extra x1,y1,x2,y2. X1 and Y1 are the top left hand of the part of the screen and x2,y2 are the points down to the bottom right-hand corner. In other words, this limits the tiles to the part of the screen between X1,Y1 and X2,Y2. This works just like the LIMIT SPRITE command.

SPOT scr,x,y,colour

Another easy one this…..SPOT is a new PLOT command, it is faster than PLOT and it allows you to plot a point on any screen in any colour. SCR can be any screen or memory bank, and COLOUR can be any colour between 0 and 15 in the present palette.

```
10 rem FILL THE SCREEN WITH DIFFERENT COLOURED DOTS

20 key off : hide on : curs off: flash off: mode 0

30 repeat

40 I=rnd(14)+1

50 X=rnd(319) : Y=rnd(199)

60 spot logic,X,Y,I

70 until mouse key
```

The next command is:

REFLECT scr1,y1,y2,scr2,y3

What this command does is produce either a ‘mirror’ effect or a ‘rippling water’ effect depending on how you use it. Now, unlike commands like screen copy and box which allow you to grab part of the screen by certain x and y coordinates. The REFLECT command only uses y coordinates. The Y coordinates mean ‘pixel lines’, if you draw a line using DRAW 0,0 to 319,0 then you’ll get a straight line across the screen. This is what I mean by pixel line. SCR1 can be either a screen or memory bank containing the picture of which you wish to reflect part of, and SCR2 is the screen to place the reflection on. To explain this command a little better I am going to use a diagram.

Y1 _____

This is the part of the picture to reflect

Y2 _____

As we can see, the REFLECT command doesn’t use X coordinates. The pixel lines start at Y co-ordinate Y1 and finish at Y co-ordinate Y2. So the part of the picture inside these lines is the part we are going to reflect. Y3 is the pixel line where we want to place the captured screen.

Y3 _____

Place the captured part of the picture here

Now, look at this example……

```
10 key off : hide on : flash off : curs off : mode 0

20 reserve as screen 7

30 load”PIC.PI1?,7

40 rem

50 rem Grab part of the picture in bank 7 between lines 50 and 100

60 reflect start(7),50,100,logic,140

70 rem The picture is placed on pixel line 140
```

If we run this program we will see that the part of the picture we captured is mirrored…ie upside down on the logical screen. What has happened is that the command has captured the reflection of the screen part.

That’s the ‘mirror image. Now let’s try the rippling water effect. Add these lines to the above routine.

```
15 logic=back

80 screen swap : wait vbl

90 goto 60
```

Note it is important that you make sure there’s enough pixel lines for the whole captured picture part or the command will squash it up to make it fit.

WASH scr,x1,y1,x2,y2

This is another version of CLS….the only difference between this and the WIPE command is that it’s used to clear only part of the screen between coordinates x1,y1,x2,y2. Note X coordinates are in steps of 16 pixels.

BLIT scr1,x1,y1,x2,y2,scr2,x3,y3

BLIT is a new faster version of screen copy. Those of you who have used SKOPY from the MISTY extension will already be familiar with this command as the parameters are the same. X1,Y1,X2,Y2 are the coordinates of the captured block of SCR1 and X3,Y3 are the coordinates of SCR2 to place it on. Again all X coordinates must be in steps of sixteen pixels.

```
10 key off : hide on : mode 0

20 rem Put a picture in bank 5

30 blit start(5),0,50,200,100,logic,32,100
```

Note that like screen copy BLIT will copy a square block onto the screen so if you are placing it over a picture then you will see the background of the picture you captured.

M BLIT scr1,x1,y1,x2,y2,logic,x3,y3

This is just like BLIT only it merges the captured block on the screen. It works like SCREEN$ but doesn't need to store the captured block in a variable first. Useful for placing part of a picture over another.

PALETTE COMMANDS

P=palt(PAL_ADDRESS)

palsplit MODE,PAL_ADDRESS,Y,YNUM,PAL_SPLIT

floodpal COLOUR

B=brightest(PAL_ADDRESS)

Let's look at these commands in turn.

P=palt (PAL_ADDRESS)

The PALT command is similar to the GET PALETTE command only it will capture the palette from an MBK file or back such as pictures, sprites, bobs, joeys, tiles, blocks, etc. It can be used instead of the routine that normally gets the palette from the sprite bank.

It seems that the command says to set a variable (P) to the palette you want STOS to take on but not so. Using this command as it's written will work just like GET PALETTE only this version will not get the palette from a normal screen. Let's say we wanted to get the palette from the sprite bank, we would use…

P=palt(start(1))

The variable can be anyone you want, ignore the number it holds. As we can see, the command sets the STOS default palette to the one in the sprite bank so we can display the sprite on screens in its own colours.

PAL_ADDRESS is as you can guess, the address of the palette to get. It can be either a screen or a memory bank.

palspilt MODE,PAL_ADDRESS,Y,YNUM,PAL_SPLIT

To the artist, the ST can be a little annoying as it can only display sixteen colours on the screen at the same time. But with this command, we can have a few different palettes on the screen at any one time. This command works by quickly splitting several palettes, in other words, switching between them so fast it looks like there are more than sixteen colours on screen at once. The only thing about it is that you can only have one different palette on any same part of the screen. Look at this example.

```
10 key off : curs off : flash off : hide : mode 0

20 reserve as screen 5 : load”pic.pi1?,5

30 load”sprites.mbk”

40 get palette(5) : SP=palt(start(1))
```

50 screen copy 5,0,0,319,50 to 0,0

60 sprite 1,100,100,1

70 palsplit 1,SP,100,199,2

80 wait key : palsplit 0,0,0,0,0

The trouble with this command as far as I've found, is that you can only display one normal picture with one palette. But you can display mbk files or banks in completely different palettes on screen at the same time. Let's look at the commands parameters.

MODE: Turns the command on and off….set to nought if off and set to one to activate it.

PAL_ADDRESS: The address of the palette to grab, which can be any screen or a bank. If it's a bank then use the START command.

Y: Which scanline to start from. To display so many palettes the command draws lines across the screen, these lines are the pixel lines I mentioned with the reflect command and we can see these lines going along the screen on an old film. If you wanted to start the pixel lines drawing from the top then we would set Y to nought.

YNUM: How many lines to draw down the screen from Y. For a full screen, we would set this to 199, the length of the screen.

PAL_SPLIT: How many palette changes to do. If you had part of a screen at the top of the screen with one palette and a sprite at the bottom half of the screen with another then we have two different palettes on screen at once, so PAL_SPLIT would be two.

In the above routine, we loaded a picture into bank five then loaded some sprites which go straight into bank one. We then put the top half of the picture on the top half of the screen then we place the sprite in the bottom half. We then use palsplit to flick in between the two palettes. Note that both parts of the screen each use a different palette. So if we move the sprite to the top half of the screen over the picture then we will see that the sprite takes the palette of the picture. So we can have one palette from a screen but loads from any mbk bank with sprites, bobs, joeys, tiles, etc. After GET PALETTE we can use the palt command as many times as we like.

floodpal COLOUR

This is an easy one. It allows you to choose a colour from the present palette and change all the other colours to that one. So if COLOUR was set to nought and that colour was black then all the other colours in the palette would become black.

B=brightest(PAL_ADDRESS)

Another easy one, it just simply finds the brightest colour in your palette. Useful if you had a game where each picture had a different palette and you wanted the text to always print in a bright colour.

10 load"picture",5

20 B=brightest(start(5))

30 pen B : print”Pen “;B;” is the brightest colour.”

FILE COMMANDS

L=dload(FILE_ADDR,ADDR,START,LENGTH)

L=dsave(FILE_ADDR,ADDR,START,LENGTH)

R=file length(FILE_ADDR)

bank load FILE_ADDR,ADDR,NUMBER

bank copy BANK1,BANK2,NUMBER

R=bank length(FILE_ADDR,NUMBER)

R=bank size(BANK,NUMBER)

Let’s look at each one in turn.

L=dload(FILE_ADDR,ADDR,START,LENGTH)The DLOAD command is a new LOAD command. It has other features as such as allowing you to choose how many bytes of a file you wish to load. The parameters are FILE_ADDR, which is the address of a variable containing the name of a file on disk.

The next parameter is ADDR which tells DLOAD where you want to load the file, either a screen or a memory bank.

DLOAD allows you to load part of a file if you so wish and therefore the START and END variables hold the byte to start loading from and the byte to stop loading at. Let’s say we wanted to load a screen into a bank. We would bear in mind that a screen is 32032 bytes long so the end of this file is 32032…the length. Look at this example program which will help to make this clearer.

```
10 key off : flash off : curs off : hide on : mode 0

20 F$=”pic.pi1?+chr$(0)

30 L=dload (varptr(F$),physic,0,32032)
```

So first we are setting the variable to hold the name of the picture file. Note the CHR$(0), this is needed by the command to read the string. We then use the VARPTR command to tell DLOAD the address of the filename so it can load it onto the physic screen starting at byte 0 and stopping at byte 32032 in the picture file. The L variable just holds the number of the bytes read.

The DLOAD command will not load a normal file from disk, it needs to load a file saved in a special format. This needs the next command.

L=dsave(FILE_ADDR,ADDR,START,END)

The DSAVE command is used to save a file that can be read by DLOAD. The parameters are the same as DLOAD only it saves the file. The good point of these two commands is that no one else can load these specially saved files and therefore they can’t use your

files in their programs. Here's a routine that will save a screen for use with the DLOAD command.

```
10 key off : hide on : curs off : flash off : mode 0

20 F$="PIC.PI1?: load F$

30 rem Save picture in special format

40 L=dsave (varptr(F$),physic,0,32032)

50 end
```

This screen can now be loaded with the DLOAD command.

R=real length(FILE_ADDR)

This command just returns the unpacked length of a pack file. R will equal the original length of the file before it was packed if it was packed. If it wasn't packed then R would equal nought.

```
10 F$="PIC.PI1?+chr$(0)

20 R=real length(varptr(F$))

30 if R<>0 then print "File not packed." else print"The original size of the file was ";R;" bytes."
```

The next command is:

bank load FILE_ADDR,BANK,NUMBER

This command allows you to load a file from a large file full of files. In other words, you can use store ten MBK files together in one bank and just take the one you want out when you need it. This is known as a file bank. A file bank is a collection of MBK or binary files stored together. This will give you a tidy disk with all the data files stored in one big one instead of loads of files scattered around the disk.

To create one of these banks we have to use the MAKEBANK program which is on the missing link disk in BAS format. Load and run it and you'll see various options. If you use the add file option you can choose a MBK or binary file to add to the file bank. Let's say you wanted to have a file bank with three MBK pictures you would click on ADD FILE with the mouse then choose a MBK picture. The program will then load it into a file bank.

Choose this option again to load more MBK pictures into the bank. You could also use the ADD DIRECTORY option to load a folder of MBK pictures……just click on the folder and then return to load all the pictures into the file bank. When you've finished click on SAVE FBANK to save the file bank. Let's say the bank had three MBK pictures in it and we wanted to load the first one from it…..here's the routine.

```
10 key off : hide on : flash off : curs off : mode 0

20 rem Reserve a bank the size of the first picture
```

```
30 reserve as data 5,12000

40 rem Load picture one into bank five

50 F$="pictures.bnk"+chr$(0): bank load varptr(F$),start(5),0

60 unpack 5
```

In a file bank, the file numbers range from nought to the number of files in the bank. In the bank the picture numbers are moved back one place so picture 1 would be picture 0, picture 2 would be picture 1, etc……so FILE_ADDR holds the address of the file banks name, BANK is the memory bank to load it into and NUMBER is the file number to load.

bank copy BANK1,BANK2,NUMBER

Supposed you made a file bank of twenty MBK pictures for use in an adventure game. You could fix the game to load extra files and call them when you need them. For example, if the game was loaded on an ST with enough memory to hold all the pictures in memory and call each one from a bank instead of a disk. On a one-meg version of the game, you could load the screens like this.

```
10 key off : hide : flash off : curs off : mode 0

20 rem Reserve bank five to the size of the file bank

30 reserve as work 5,40000

40 bload "pics.bnk",start(5)

50 rem Put the first picture on the screen

60 reserve as data 6,2580 : bank copy start(5),start(6),0

70 unpack 6
```

So BANK1 is the address of the file bank, BANK2 is the address of the bank you've copied the picture into and NUMBER is the number of the picture to copy ranging from 0 to the number of pics in the bank. In this example, we must use BLOAD instead of BANK LOAD.

R=bank length(FILE_ADDR,NUMBER)

Simply returns the length of file number NUMBER in FILE_ADDR. Useful for finding out the length of the file in the bank so you can reserve a bank for it. R equals the size of file NUMBER in FILE_ADDR.

```
10 F$="pics.bnk"+chr$(0)

20 R=bank length(varptr(F$),1)

30 reserve as data 5,R

40 print"Picture 1 is ";R;" bytes long."
```

R=bank size(ADR,NUMBER)

The last command checks the length of a file in a file bank on disk but this command checks for the length of a file in a file bank in memory. ADR is the bank containing the file bank and NUMBER is the number of the file you want to check.

```
10 R=bank size(start(5),0)

20 print “Picture 1 is “;R;” bytes long.”
```

SOUND COMMANDS

digiplay MODE,ADDR,SIZE or SAMPLE NUMBER,FREQ,LOOP

samsign ADDR,SIZE

R=musauto(ADDR,NUMBER,SIZE)

musplay ADDR,NUM,OFFSET

The missing link has a nice selection of sound commands. Here is the first one.

digiplay MODE, ADDR, SIZE or SAMPLE NUMBER, FREQ, LOOP

The digiplay command allows you to play a sample. Unlike the STOS maestro extension the digiplay takes up less processor time giving your game more speed. Here's what each parameter means…

MODE: Bit pointless this one – it just turns the command on and off, if it’s set to one then it’s on, if it's off it's set to nought. So, you pass MODE as nought then the command is ignored. Might as well keep it set to one.

ADDR: The address of the raw sample. It can be a memory bank or even a screen. Note if it's in a bank then you must use the start command.

SIZE: The actual size of the sample in bytes. You can find out the size by listing a directory that gives file sizes. Note this can also be a sample number, more on this later.

FREQ: The playback speed of the sample between 3-25 Khz.

LOOP: if you want to play a sample once then set this to nought. If you want the sample to keep playing then set it to one.

The sample must be loaded with the bload command for digiplay to understand it. Here is an example that plays a sample 3000 bytes long in a loop from bank five at speed 10.

```
10 reserve as work 5,3000 : bload”sample.sam”,5

20 digiplay 1,start(5),3000,10,1

30 wait key

40 digiplay 0,0,0,0,0
```

Using line 40 will stop the sample from playing in a mind-numbing loop.

This example will play one raw sample. But with Maestro we can hold a few samples in a bank and play one like this

samplay 2 : rem plays sample two

This is possible with digiplay, load up the MAKE program and use it to make a digibank. To do this just load your samples into one after the other then choose the save digibank option. You can load up to 50 samples in any one bank. Now to load and play it….

10 reserve as work 5,30000 : bload”sambank.mbk”,5

20 digiplay 1,start(5),2,10,1

This is why the third variable has two different meanings. If it’s more than 50 then the command assumes you want to play a raw sample. But if it's less than 50 then it assumes you want to play a sample from a digibank held in a memory bank. Note that unlike maestro the sample number starts from nought….so sample 1 would become sample 0, sample 2 would become sample 1, etc….the above example plays sample three from the digibank in memory bank five.

samsign ADDR,SIZE

I don’t know much about signed and unsigned samples only that sometimes a sample will sound distorted when played with digiplay. The samsign will sign the sample if it's unsigned or visa versa.

samsign start(5),3000

digiplay 1,start(5),3000,10,0

So ADDR is the address of the sample you wish to sign or unsign depending on its status. And SIZE is the length of the sample.

R=musauto (ADDR,NUMBER,SIZE)

A lot of people including myself can’t stand the music created with the STOS music accessory. Instead, we use other chip music such as mad max. There are various music creator programs around such as the Megatiser which allow you to create your own xbios chip tunes.

The musauto command will automatically play one of twenty-one different kinds of chip music. The parameters are:

ADDR: The address of the music, usually a bank

NUMBER: The musauto command checks for the music to see how many tunes are in it. Some mad max music can have two tunes stored together like a stacked bank. Pass this parameter as the number of the music you want to play starting from one.

SIZE: The length of the music.

The command works by looking at the offset of the music and setting itself up to play that kind of music. For example…

reserve as work 10,5000 : bload”madmax.mus”,10

R=musauto(start(10),1,5000)

This plays mad max music on interrupt. Note how there's no setting it to the offset of mad max music as the command works out what music type it is. To stop the music playing just use….

R=musauto(start(5),0,5000)

Passing NUMBER as 0 will stop the music. There is a list of what tunes musauto will play in the link document.

musplay ADDR, NUMBER, OFFSET

Although musauto can play up to 21 kinds of music, there will be the odd tune it doesn't recognise because it can't work out the offset number that starts it playing. Musplay will play other kinds of chip music if you know the offset.

musplay start(5),1,1

Mad Max music has an offset number of one. So the example plays tune one of mad max music.

JOYSTICK COMMANDS

D=P JOY(N)

P STOP

P ON

P UP(N)

P DOWN(N)

P LEFT(N)

P RIGHT(N)

The joystick commands in the missing link are just the same as the default STOS ones only they have a couple of advantages. Two commands allow you to turn the joysticks on and off. These are P STOP and P ON. I suppose a command called P OFF would have us all giggling.

10 repeat

20 if p up(1)=true then y=y-2

30 if J=1 then p stop else p on

40 until p fire(1)

So, in a normal listing using commands like JUP and JDOWN you would have to have a variable telling STOS to ignore the lines that check for the joystick where all you have to do here is to turn the joystick off. So to look at this command again the format is

p stop (Turns off the joystick ports)

Before you can use these joystick commands you have to turn them off.

p on

So you can use P ON to activate the ports to use these joysticks and P STOP to turn them off.

Another advantage of these commands is to allow you to read both ports. You may know that you can have two joysticks connected to your ST. One in the joystick port (port 1) which is the second port and the mouse port (port 0) which is the first one. Of course, if you check for a joystick in the mouse port then that means you can't use the mouse.

Let's look at the other commands in more detail.

P UP(n)

This is just like the JUP command in STOS except that it has an extra parameter. The parameter N is the number of the port you want to check. Let's see this example.

```
10 p on : rem FIRST TURN THE PORTS ON

20 repeat

30 if p up(1) then y1=y1-4

40 if p up(0) then y0=y0-4

50 until inkey$=" "

60 p stop
```

In this loop, we have two variables called Y0 and Y1 which hold the Y coordinates of two sprites on screen. Line 30 checks to see if the joystick in port one (joystick port) has been pushed up and if so, take four away from the Y1 variable. Line 40 does the same, only for port nought (mouse port). When the space bar is pressed the joysticks are turned off. To explain it all. Each of these commands is the same as the joystick commands only that they allow you to check the joysticks in both ports rather than just the normal port one. Here are some examples.

```
if p down(0) then y0=y0+4 : rem Check if the first joystick pulled down

if p left(0) then x0=x0-4 : rem Check if the first joystick pushed left

if p right(0) then x0=x0+4 : rem Check if the first joystick pushed right

if p up(0) then y0=y0-4 : rem Check if the first joystick pushed up

if p fire(0) then F=1 : rem Check if fire button pressed
```

Put these lines in a loop and it allows you to check if the joystick in port nought (the mouse port) has been accessed. Changing the noughts to ones allows you to simply check the joystick port (port one). Note the P FIRE command, this allows you to check if the fire buttons have been pressed.

Note how these commands are listed at the start as….

D=j up(n)

When the joystick is moved, the variable D equals one of two values.

0 : rem joystick hasn't been touched.

1 : rem Joystick has been moved.

So D or whatever variable you wish to use will either equal one if the joystick has just been moved and nought if it hasn’t.

D=P JOY(n)

This allows you to check if the joystick has been moved like the JOY command in STOS. D equals nought if it's left alone and 1 if it’s been moved.

Finally, an example of the commands working together.

```
5 F0=p fire(0) : F1=p fire(1) :rem Variables for fire buttons

10 X1=40 : X2=40 : Y1=60 : Y2=60 : P ON : rem Set up variables and turn on.

20 REPEAT

30 U0=p up(0) : D0=p down(0) : L0=p left(0) : R0=p right(0) : rem Variables for joystick in port nought (mouse port)

40 U1=p up(0) : D1=p down(1) : L1=p left(1) : R1=p right(1) : rem Variables for joytick in port 1 (joystick port).

50 A=p joy(0) : B=p joy(1) : rem Check if joysticks moving or still

60 if U0=1 then Y1=Y1-4 : rem Joystick 0 pushed up

70 if D0=1 then Y1=Y1+4 : rem Joystick 0 pulled down

80 if L0=1 then X1=X1-4 : rem Joystick 0 pushed left

90 if R0=1 then X1=X1+4 : rem Joystick 0 pushed right

100 if U1=1 then Y2=Y2-4 : rem Joystick 1 pushed up

110 if D1=1 then Y2=Y2+4 : rem Joystick 1 pulled down

120 if L1=1 then X2=X2-4 : rem Joystick 1 pushed left

130 if R1=1 then X2=X2+4 : rem Joystick 1 pushed right

140 if A=0 then home : print”Joystick 0 is not being moved”

150 if B=0 then locate 0,1 : print”Joystick 1 is not being moved”

160 sprite 1,X1,Y1,1 : wait vbl : sprite 2,X2,Y2,2 : wait vbl

170 until F0=1 or F2=1
```

The above routine will allow you to use both joysticks to move two sprites at the same time. It also lets you know if the joysticks are being used. And finally, the last line ends the routine if one of the fire buttons is pressed on either joystick.

MISCELLANEOUS COMMANDS

I=depack(ADDRESS)

D=compstate

relocate PROG_ADDRESS

R=boundary (N)

R=overlap (x1,y1,x2,y2,wid1,hg1,wd2,hg2)

Right, the first command.

The DEPACK command does as it says. It depacks a data file. In other words, it allows you to unpack or un-compress a data file. It works with various packers including Atomik 3.5 and Ice Pack 2.4. The format is.

L=depack (ADR)

ADR can either be a screen or memory bank. L holds the length in bytes of the file size and will equal nought if the file isn't packed.

```
10 reserve as work 5,free-10000: load"music.mod",5

20 L=depack(start(5))

30 if L=0 then print"File is not packed.":stop

40 if L<>0 then print"The file size is";L
```

When you pack a data file, always reserve your bank to the original size of the file before it was packed because the command unpacks the file on top of itself. It works well with MODS and chip music such as mad max.

D=compstate

This will return either one or nought depending on the state of your program. If you run it from within STOS basic, IE, interpreter mode it will equal 0. But if the listing is compiled it will equal one. Its main use is for checking if your program is compiled. The advantage is that because the compiler speeds variables up (adding, subtracting, etc). It's needed to set the program to run at certain speeds so it doesn't run too fast when complied. Take this listing for example.

```
10 if compstate=0 then NUMBER=1000

20 if compstate=1 then NUMBER=3000

30 repeat

40 dec NUMBER : home : print NUMBER

50 until NUMBER=0
```

The use of compstate in this listing means that the variable NUMBER will take the same amount of time to count down in either interpreter mode or compiled mode. For this to

happen we have to make the variable a higher number if the program has complied or it will count down much faster than we need.

The next command is:

relocate PROG_ADDR

Normally when you load a machine code program into STOS, you can only load it into a memory bank otherwise the CALL command won't work. The RELOCATE command sets up CALL to call the routine from something other than a bank. So you can run it from a screen address if you want.

10 bload"SPEED.PRG",back

20 relocate back

30 call back

It's important in this case to bload it rather than a normal load.

R=boundary (N)

The ST has a screen problem. Parts of the screen have to be copied on a sixteen-pixel boundary. In other words, graphics have to be screen copied across the screen in a step of 16 pixels. For example.

screen copy 5,16,30 to physic,32,30

So X co-ordinates have to be in steps of 16 - (0,16,32,48,64 etc).

So, if we wanted to take a number and print the nearest sixteen-pixel step we use this command. Take this listing.

10 X=11

20 N=boundary (X)

30 print "The nearest 16-pixel boundary is ";N

The command has rounded the value of X to the nearest pixel which is sixteen because 11 is the nearest to 16 than 0 is. Try changing X to other numbers and see what happens. For example, if X=25 then N would equal 32 because 25 is nearer to 32 than 16. N is the rounded-up figure.

R=overlap (x1,y1,x2,y2,wd1,hg1,wd2,hg2)

Overlap is a collision detection command. It allows you to check if part of the screen collides with another. It is useful to check if a bullet or a sprite has reached a certain part of the screen.

X1 and Y1 are the top left-hand co-ordinates of the first part of the screen you want to check, X2 and Y2 are the top left-hand corners of the second part of the screen to check, WD1 and HD1 are the width and height of the first part of the screen and WD2 and HG2 are the second part. So let's say we wanted to check if a sprite had entered the top left-hand corner of the screen.

```
10 XSP=x sprite(1) : YSP=y sprite(1)

20 A=overlap (XSP,YSP,0,0,16,16,16,16)

30 if A then boom : stop

40 if jleft then XSP=XSP-4

50 if jright then XSP=XSP+4

60 if jup then YSP=YSP-4

70 if jdown then YSP=YSP+4

80 sprite 1,XSP,YSP,1 : wait vbl

90 goto 10
```

So first the variables XSP and YSP hold the X and Y coordinates of the sprite and this is entered in OVERLAP as the top left-hand coordinates of the first block to check…ie: the top left-hand corner of the sprite. We're checking the top left-hand corner of the screen for the sprite entering it so the coordinates are 0,0 the top left-hand corner of the second part (collision block). Next, we have the size of the sprite in pixels which is 16 by 16 pixels. And finally, the size of the on-screen block is the same (16 by 16 pixels). When the sprite enters this invisible block then A equals other than nought and a collision is detected then the program stops.

If you're using sprites then the hot spot must be on the top left-hand corner of the sprite unless you're using bobs then the hot spot is always in that place.

COMMANDS FOR REGISTERED USERS

If you have a full registered copy of Missing Link then you'll have access to these commands as well.

MANY BOB

The many bob command, as it says, will put many bobs on the screen. It's much faster than using the normal bob command to put more than one bob on screen. Like the normal bob command, there is a version of the command to set a clipping zone for the many bobs, which is.

```
many bob X1,Y1,X2,Y2,0,0,0,0,0,1
```

All those noughts don't mean anything. Something Top Notch never bothered with.

The next command draws the bobs.

```
many bob SCR,ADR,IMADR,XADR,YADR,STADR,XOFF,YOFF,NUM,0
```

Confused? Well, it's not as bad as it looks. Let's go step by step through each variable.

SCR: The screen where the bobs are being displayed. This can be back, physic, or logic.

ADR: This is the bank containing the bobs (bob bank).

IMADR: This is a pointer to an array holding the image numbers of each bob.

XADR: This is a pointer to an array holding the X coordinates of each bob.

YADR: This is a pointer to an array holding the Y coordinates of each bob.

STADR: An array of numbers for each bob. If the number is one then the bob is displayed, otherwise if nought it isn't.

XOFF: A variable containing how many X pixels to move a bob.

YOFF: A variable containing how many Y pixels to move a bob.

NUM:- Number of bobs to put on screen.

Let's try to display, animate, and move three bobs at once.

```
10 key off : hide on : curs off : flash off : mode 0
```

```
20 dim IMAGE(3),XBOB(3),YBOB(3),STATUS(3)
```

Now let's set up the arrays. First the image numbers for each bob.

```
30 IMAGE(1)=0 : IMAGE(2)=1 : IMAGE(3)=2
```

Now, the X coordinates of each bob.

```
40 XBOB(1)=16 : XBOB(2)=32 : XBOB(3)=48
```

Same with the Y co-ordinates.

```
50 YBOB(1)=20 : YBOB(2)=30 : YBOB(3)=40
```

Now the status of each bob, (1) means the bob is displayed, and (2) means it isn't displayed.

```
60 STATUS(1)=1 : STATUS(2)=1 : STATUS(3)=1
```

Now let's display them on-screen……

```
70 many bob logic,start(4), varptr(IMAGE(0)), varptr(XBOB(0)), varptr(YBOB(0)), varptr(STATUS(0)),X,Y,3,0
```

Run this program and you'll see three bobs displayed on screen all with different image numbers and coordinates. Now change the contents of status two to contain 0 instead of one by amending line 60.

Run the program and you'll see the middle bob has gone. This is because we've told the program not to display that one. This is useful for getting rid of an alien that's been shot, just change its status to 0.

Animating the bobs is easy, just set the image numbers to the right frames and just go back to the line. For example.

```
100 many bob logic, start(1), varptr(IMAGE(0)) etc….
```

```
110 wait 10: IMAGE(1)=4 : IMAGE(2)=8 : IMAGE(3)=12 :GOTO 100
```

Moving is a bit strange, you have to set the bob to move in the other direction you want it to. For example, this line:

120 if jright then X=X-4

Will cause the bob to move right, and plus-ing it will cause it to go left.

Try playing about with this command and you'll find it working for you sooner or later. Errm, right, what's next?

MANY JOEY

This is just the same as many bob but it's to draw loads of joeys. However, there is an extra array that holds the colour of each bob.

MANY BULLET

Draws loads of bullets on the screen, the same as many joey but the IMAGE variable is not used.

H=HERZ

This tells you what screen frequency your ST is running under, either 50, 60, or 70 for mono monitors. H holds the value.

SET HERTZ

Set the hertz rate to another frequency, 50 or 60, but 70 only on a mono monitor, for example.

10 wait vbl : set hertz 60

Finally, the last command:

A=mostly harmless (1,2,3,4,5)

If you have an unregistered version of the missing link you get a message popping up regularly telling you to register. This command comes from the registered version and stops the message.

That's the end of this (very long) tutorial. I didn't cover every single command but there is enough here to be getting on with.

HOW TO PROGRAM YOUR OWN FILE SELECTOR

The STOS file selector looks nice but wouldn't it be nicer to code your own file selector. This is the code for the file selector in Grafix Art, my art program.

```
2240 rem FILE SELECTOR

2250 wipe S5 : unpack S2+33020,S5 : get palette (5) : wait vbl : A=0 : PRE_Y=0 : PR$="" 2260 fastcopy S5,back : fastcopy S5,logic : K=1

2270 if LO=1 then T$="LOAD A PICTURE" : pen 15 : locate 14,23 : print T$

2280 if SA=1 then T$="SAVE A PICTURE" : pen 15 : locate 14,23 : print T$

2290 T$=right$(PATH$,4) : locate 24,7 : pen 15 : print T$ : fastcopy logic,back

2300 rem

2310 blit S5,96,52,160,64,back,96,52 : blit S5,96,52,160,64,logic,96,52 : wash back,112,72,224,152

2320 if drive=0 then logic=S3 : ink 15 : bar 2,2 to 30,30 : paper 15 : pen 0 : locate 1,1 : print "A" : logic=physic : ppsc S3,logic,3,7,20,15,107,55,4 : paper 0 2330 if drive=1 then logic=S3 : ink 15 : bar 2,2 to 30,30 : paper 15 : pen 0 : locate 1,1 : print "B" : logic=physic : ppsc S3,logic,3,7,20,15,132,55,4 : paper 0 2340 fastcopy logic,back

2350 dir$="\" : DD$=""

2360 blit S5,16,32,208,41,back,16,32 : blit S5,16,32,208,41,logic,16,32

2370 for X=0 to 500 : FILE$(X)="" : IDEN(X)=0 : F_NAME$(X)="" : E$(X)="" : POS(X)=0 : next X 2380 FILE=0 : FI=8 : POS=0 : S=0 : wash back,112,72,224,152 : wash logic,112,72,224,152 2390 rem GET FOLDERS

2400 P$="*.*" : N$=dir first$(P$,-1)

2410 if mid$(N$,43,1)="8" then N$=dir next$

2420 if left$(N$,1)="." then N$=dir next$ 2430 if left$(N$,2)=".." then N$=dir next$

2440 D$=mid$(N$,43,2) : if D$="16" then inc FILE : F_NAME$(FILE)=left$(N$,12) : gosub 2740 : FILE$(FILE)=left$(N$,12) : IDEN(FILE)=1 2450 repeat

2460 N$=dir next$

2470 D$=mid$(N$,43,2) : if D$="16" then inc FILE : F_NAME$(FILE)=left$(N$,12) : gosub 2740 : FILE$(FILE)=left$(N$,12) : IDEN(FILE)=1 2480 until N$=""

2490 rem GET FILES

2500 P$=PATH$ : N$=dir first$(P$,-1)

2510 D$=mid$(N$,43,2) : if D$<>"16" then inc FILE : F_NAME$(FILE)=left$(N$,12) : gosub 2740 : FILE$(FILE)=left$(N$,12) : IDEN(FILE)=2 2520 repeat

2530 N$=dir next$
```

2540 D$=mid$(N$,43,2) : if D$<>”16” then inc FILE : F_NAME$(FILE)=left$(N$,12) : gosub 2740 : FILE$(FILE)=left$(N$,12) : IDEN(FILE)=2 2550 until N$=””

2560 rem NOW LIST THEM IN THE FILE SELECTOR

2570 for Y=1 to FILE

2580 T$=FILE$(Y+S)+chr$(0) : text back,0,varptr(T$),15,Y+FI

2590 if IDEN(Y+S)=1 then T$=”*”+chr$(0) : text back,0,varptr(T$),14,Y+FI

2600 if POS<10 then inc POS : POS(POS)=POS

2610 if Y=10 then goto 2630

2620 next Y

2630 blit back,112,72,224,152,logic,112,72 : paper 0

2640 rem SET ZONES

2650 set zone 1,233,68 to 255,84

2660 set zone 2,233,140 to 255,154

2670 set zone 3,112,70 to 224,152

2680 set zone 4,167,156 to 228,170

2690 set zone 5,155,49 to 176,61

2700 set zone 6,106,157 to 163,168

2705 set zone 7,105,53 to 120,64

2710 set zone 8,131,47 to 150,60

2720 set zone 9,103,32 to 200,46 : show on 2725 if LO=1 then reset zone 9

2726 goto 2860

2740 rem SPACE FILE NAMES FOR FILE SELECTOR 2750 D=instr(N$,”.”) : if D=0 then return

2760 A$=left$(N$,D-1) : E$(FILE)=mid$(N$,D+1,3) : E$=mid$(N$,D+1,3) 2770 if len(A$)=1 then SP=8

2780 if len(A$)=2 then SP=7

2790 if len(A$)=3 then SP=6

2800 if len(A$)=4 then SP=5

2810 if len(A$)=5 then SP=4

2820 if len(A$)=6 then SP=3

2830 if len(A$)=7 then SP=2

2840 if len(A$)=8 then SP=1

2850 N$="" : N$=A$+space$(SP-1)+"."+E$: return 2860 rem GET NAME AND POSITION OF LAST FILE SELECTED 2870 if A>7 then PRE_Y=A : PR$=FILE$(A-8+S)

2880 rem WAIT FOR CHOICE

2890 repeat : M=mouse key : CH=zone(0) : wait vbl : until M and CH<>0 2900 rem CHOICE SELECTED

2910 if M=1 and CH=1 and S>0 then dec S : dec POS

2920 if M=1 and CH=2 and S<FILE-11 then inc S : inc POS : POS(POS)=POS 2930 if M=1 and CH=3 then A=ytext(y mouse) : H=POS(A-8+S) : goto 3090 2940 if M=1 and CH=4 and FF$<>"" then goto 3140

2950 if M=1 and inkey$=chr$(13) and FF$<>"" then goto 3140

2960 if M=1 and CH=4 and FF$="" then goto 2880

2970 if M=1 and inkey$=chr$(13) and FF$="" then goto 2850

2980 if M=1 and CH=5 and FOLD=1 then hide on : goto 3350 2990 if M=1 and CH=5 and FOLD=0 then hide on : goto 2300 3000 if M=1 and CH=5 and FOLD=0 then goto 2890

3010 if M=1 and CH=6 then hide on : cls : wipe S5 : wipe S3 : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220 3020 if M=1 and CH=8 and drvmap<3 then goto 2560

3030 if M=1 and CH=7 then hide on : drive$="A:" : goto 2310

3040 if M=1 and CH=8 then hide on : drive$="B:" : goto 2310

3050 if M=1 and CH=9 then goto 3400

3060 rem UPDATE SCREEN

3070 wash back,112,72,224,152

3080 goto 2560

3090 rem HIGHLIGHT THE OPTION

3100 paper 0 : pen 15 : locate 15,PRE_Y : print PR$

3110 inverse on : locate 15,A : F$=FILE$(A-8+S) : FF$=F_NAME$(A-8+S) : pen 15 : print F$: inverse off

3120 pen 15 : locate 13,4 : A$=left$(F$,8) : print A$: locate 21,4 : A$=" "+right$(F$,3) : print A$: wait 10 3130 goto 2870

3140 rem IF FILE IS A FOLDER THEN OPEN IT AND LIST FILES

3150 for X=1 to FILE

3160 K$=F_NAME$(X) : if K$=FF$ and IDEN(X)=1 then FOLD=1 : P$="" : goto 3300

3170 next X : FOLD=0

3180 if SA=1 then goto 3550

3190 if LO=1 and PATH$=”*.PI1” then hide on : wait 10 : load FF$: SCR$(SC)=”” : SCR$(SC)=screen$(logic,0,0 to 330,200) : gosub 3220 : wait 100 : cls : wipe S3 : wipe S5 : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220 3200 if LO=1 and PATH$=”*.NEO” then hide on : wait 10 : load FF$: SCR$(SC)=”” : SCR$(SC)=screen$(logic,0,0 to 330,200) : gosub 3220 : wait 100 : cls : wipe S3 : wipe S5 : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220 3210 if LO=1 and PATH$=”*.GRX” then hide on : goto 3260

3220 for COCO=0 to 15 : C0L$(SC,COCO)=”” : next COCO

3230 for COCO=0 to 15 : C0L$(SC,COCO)=hex$(colour(COCO)) : next COCO 3240 return

3260 rem

3270 reserve as work 10,20000 : fill start(10) to start(10)+length(10),0

3280 bload FF$,10 : crack unpac start(10),logic,0 : wait 100 : SCR$(SC)=”” : SCR$(SC)=screen$(logic,0,0 to 330,200) : erase 10 : gosub 3220 3290 cls : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220

3300 rem OPEN THE FOLDER

3310 gosub 3390

3320 dir$=”\”+DD$

3330 wash back,112,72,224,152

3340 P$=”*.*” : goto 2360

3350 rem MOVE BACK TO PREVIOUS PATH

3360 dir$=”\” : DD$=””

3370 wash back,112,72,224,152

3380 FOLD=0 : goto 2360

3390 DD$=DD$+FF$+”\”-“ “ : return

3400 rem ENTER A SAVE FILE NAME

3410 reset zone : set zone 1,167,160 to 228,174

3415 locate 13,4 : print space$(12)

3420 clear key : XC=13 : XSP=104 : locate XC,4 : FF$=”” : T$=”ENTER NAME OF PICTURE” : pen 15 : locate 10,23 : print T$: sprite 1,XSP,32,2 : caps lock on 3430 K$=inkey$

3440 rem CHECK FOR RETURN KEY

3450 if K$=chr$(13) then goto 3545

```
3460 if mouse key=1 and zone(0)=1 then goto 3545

3470 rem CHECK FOR BACKSPACE KEY

3480 if K$=chr$(8) and XC>12 and XSP>104 then locate XC,4 : print " "; : dec XC : locate XC,4 : XSP=XSP-8 : sprite 1,XSP,32,2

3490 if XC=12 then XC=13

3500 rem CHECK FOR NORMAL CHARACTERS

3510 if K$="." then XC=22 : XSP=176 : sprite 1,XSP,32,2 : locate XC,4 : goto 3540

3520 if asc(K$)>31 and XC=21 then XSP=XSP+8 : sprite 1,XSP,32,2 : inc XC : locate XC,4 : print K$; : goto 3540

3530 if asc(K$)>31 and XC<25 then XSP=XSP+8 : sprite 1,XSP,32,2 : print K$; : inc XC

3540 K$="" : locate XC,4 : goto 3430

3545 for X=13 to XC : FF$=FF$+chr$(scrn(X,4)) : next X

3550 if len(FF$)=0 or len(FF$)=1 then sprite off : paper 0 : locate 10,23 : print space$(23) : wait 10 : goto 2640 3560 F=instr(FF$,".") : if F then FF$=left$(FF$,F-1) : hide on : wait 10

3565 if not(F) then FF$=left$(FF$,8)-" " : hide on : wait 10

3570 if PATH$="*.PI1" then cls : goto 3680

3580 if PATH$="*.NEO" then cls : goto 3700

3590 if PATH$="*.GRX" then FF$=left$(FF$,8)-" "

3600 rem SAVE A PACKED SCREEN

3610 cls : reserve as work 10,20000 : fill start(10) to start(10)+length(10),0 3620 for X=0 to 15 : colour X,val(C0L$(SC,X)) : next X

3630 opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back

3640 D=crack pac(logic,start(10))

3650 bsave FF$+".GRX",start(10) to start(10)+length(10) : erase 10 : gosub 3220 3660 cls : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220 3670 rem

3680 for X=0 to 15 : colour X,val(C0L$(SC,X)) : next X

3690 opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : save FF$+".PI1" : cls : wipe S3 : wipe S5 : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220 3700 for X=0 to 15 : colour X,val(C0L$(SC,X)) : next X

3710 opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : save FF$+".NEO" : cls : wipe S3 : wipe S5 : unpack S2,S5 : floodpal 0 : wait vbl : fastcopy S7,S3 : goto 220
```

DEMO PROGRAMMING IN STOS

A team of programmers (myself included) got together to create what is known as the STOSSER Demo. In this article I will show you how you can write a demo without the complicated stuff the big boys can do. All you need is to produce something that looks nice. The examples below are my contributions to the STOSSER demo and while they are simple programming, they actually look quite good.

This first Routine has the words SILLY SOFTWARE dancing in the background along with some VU bars moving in time with the music. It also has a star background and the words SILLY SOFTWARE shimmering underneath the VU bars. Looks like a lot of hard work but in fact it isn't. Here's the routine.

```
10 key off : hide on : click off : flash off : curs off : mode 0

15 LE=12800 : F$="circus.MUS"

16 remreserve as work 2,LE : bload F$,2

20 P=palt(start(3)) : logic=back : wipe stars on : set stars 200,2,0,0,330,200,1,15

50 I=depack(start(2)) : A=musauto(start(2),1,LE) : S4=start(4) : S3=start(3)

60 rem START

70 go stars 0,1,logic

80 A=psg(8) : B=psg(9) : C=psg(10)

90 bob logic,S4,A,128,80,0 : bob logic,S4,B,152,80,0 : bob logic,S4,C,176,80,0

95 bob logic,S3,0,48,40-A,0 : bob logic,S3,1,64,40-B,0 : bob logic,S3,2,80,40-C,0 :
bob logic,S3,2,96,40-A,0 : bob logic,S3,3,112,40-B,0 96 bob logic,S3,0,144,40-A,0 :
bob logic,S3,4,160,40-B,0 : bob logic,S3,5,176,40-C,0 : bob logic,S3,6,192,40-A,0 :
bob logic,S3,7,208,40-B,0 97 bob logic,S3,8,224,40-A,0 : bob logic,S3,9,240,40-B,0 :
bob logic,S3,10,256,40-C,0

105 Z$=inkey$ : if Z$=" " then A=musauto(0,0,0) : fade 3 : wait 40 : default : end

106 reflect logic,20,70,logic,150

110 screen swap : wait vbl : goto 60
```

That's all it is - the words SILLY SOFTWARE are put on screen as bobs letter by letter and the STOS "psg" command moves the letters up and down in time with the chip tune playing. Creating the shimmering affect is used with just one command, "reflect" from the ML extension.

The routine below simply moves a multicoloured box around the screen whilst using VU bars to move in time with the music. Then I just add some sentences in data statements that appear at the bottom of the screen.

```
10 curs off : palette 0,0,0,0,0,0,0,0,0,0,0,0 : hide on : key off : mode 0 : click off :
disable mouse : break off 15 reserve as screen 4 : S4=start(4) : wipe S4
```

```
20 palette
$0,$777,$577,$377,$177,$375,$573,$771,$750,$730,$700,$702,$703,$705,$727,$7
47

40 logic=back : A=0 : C=1 : dreg(0)=1 : call start(5) : loke $4DA,start(5)+8 :
timer=0

50 X1=160+sin(A/21.6)*45

60 X2=160+sin(A/16.6)*45

70 Y1=58+cos(A/22.6)*40

80 Y2=58+cos(A/22.6)*40

90 ink C : C=(C mod 14)+2

95 CH1=psg(8) : CH2=psg(9) : CH3=psg(10)

100 fastcopy S4,back : bar X1,Y1 to X2,Y2-CH1

101 ink C : C=(C mod 14)+2

102 bar X1,Y1+40 to X2,Y2+40-CH2

103 ink C : C=(C mod 14)+2

104 bar X1,Y1+80 to X2,Y2+80-CH3

105 if TT=72 then TT=0 : restore 140

106 if timer>150 then read T$ : wipe S4 : logic=4 : locate 0,23 : centre T$ :
logic=back : timer=0 : inc TT

110 screen swap

120 wait vbl : inc A

125 if inkey$=" " then loke $4DA,0 : call start(5)+4 : goto 65000

130 goto 50

140 data “Hello, this is Deano. Welcome to my Demo for the Stosser Demo”

150 data “Blah Blah Blah”
```

And finally, here is a similar routine that moves and draws some colourful lines along a sin wave.

```
10 curs off : hide on : key off : mode 0

20 palette
$0,$11,$577,$377,$177,$375,$573,$771,$750,$730,$700,$702,$703,$705,$727,$74
7

30 def scroll 1,64,16 to 256,200,0,8

40 logic=back : A=0 : C=1
```

```
50 X1=160+sin(A/21.6)*95
60 X2=160+sin(A/16.6)*95
70 Y1=58+cos(A/22.6)*40
80 Y2=58+cos(A/12.6)*40
90 ink C : C=(C mod 14)+2
100 draw X1,Y1 to X2,Y2
110 scroll 1 : screen swap
120 wait vbl : inc A
130 goto 50
```

So, this is the source for my two contributions. You can download the Stosser Demo and other STOS demos from

https://www.exxosforum.co.uk/atari/STOS/DEMOS/DEMOS.htm

PROGRAMMING AN ART PACKAGE

Ever fancied writing your own art package? Well in this tutorial I am going to show you how to program a few options that you would normally find in an art package. I have written a full art package called Grafix Art and the routines printed here are taken straight from it.

First of all, this is the routine for the normal freehand draw tool.

```
4410 rem DRAW

4420 hide on : off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : show on : K=1

4430 set mark 1,1

4440 wait 10 : repeat

4450 X=x mouse : Y=y mouse

4460 if hardkey=97 then hide : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : fastcopy logic,back : show

4470 until mouse key=1 or mouse key=2

4480 hide on : SCR$(SC)=screen$(logic,0,0 to 330,200)

4490 if mouse key=2 then hide on : wait 10 : goto 230

4500 if mouse key=1 then polymark X,Y

4520 repeat

4540 X=x mouse : Y=y mouse

4550 if mouse key=1 then polyline to X,Y

4560 until mouse key=0

4570 fastcopy logic,back

4580 show on : goto 4440
```

In other STOS art packages, people have tried using the PLOT command for this routine. However, this causes the line to draw broken. What happens here is we use polymarks to make the lines. The opaque command is from the EXTRA extension and what it does is places an image in a string put there using SCREEN$ onto the screen as a normal screen copy and not merged. In an art package, we have workscreens to place our pictures in so we can design so many different pictures at a time and flick through them. The routine starts by waiting for the user to click the left mouse button before it will draw a polymark. The right mouse button allows you to abort the operation.

This next routine will draw a continuous line.

```
4590 rem LINE
```

```
4600 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : K=1 : set line val(LTYPE$),L_SIZE,0,0 : show on

4610 clear key : wait 10 : repeat

4620 X=x mouse : Y=y mouse : M=mouse key : K$=inkey$

4630 if asc(K$)=0 and scancode=97 then hide : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back : show

4640 wait vbl : until M

4650 if M=2 then hide on : wait 10 : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : cls : goto 200

4660 Y1=Y : X1=X : hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200)

4670 logic=back : wait 10 : repeat

4680 opaque screen back,0,0,SCR$(SC) : polyline X,Y to X1,Y1

4690 X1=x mouse : Y1=y mouse

4700 if mouse key=2 then logic=physic : wait 10 : cls : goto 200

4710 screen swap : wait vbl

4720 until mouse key=1

4730 logic=physic

4740 show on : goto 4610
```

You'll notice in these routines that we have implanted an UNDO feature. This means that if you are not happy with the last thing you have done you can erase it from the screen before you carry on. Note that the routine will put the current picture into a screen with SCREEN$. This is your present work screen.

Drawing a box is easy as well, here is the routine.

```
4750 rem BOX

4760 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : K=1 : set line val(LTYPE$),L_SIZE,0,0 : show on

4770 wait 10 : repeat

4780 X=x mouse : Y=y mouse : wait vbl

4790 if hardkey=97 then hide : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back : show

4800 M=mouse key : until M

4810 Y1=Y : X1=X : hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200)

4820 if M=2 then hide on : wait 10 : cls : goto 200
```

```
4830 logic=back : repeat
4840 opaque screen back,0,0,SCR$(SC) : box X,Y to X1,Y1
4850 X1=x mouse : Y1=y mouse
4860 screen swap : wait vbl
4870 until mouse key
4880 if mouse key=2 then logic=physic : wait 10 : cls : goto 200
4890 logic=physic : fastcopy logic,back
4900 show on : goto 4770
```

You can also alter the routine to draw a rounded box just by changing the command BOX to RBOX.

Something a little more interesting……a triangle.

```
4910 rem TRIANGLE SHAPE
4920 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : K=1 : set line val(LTYPE$),L_SIZE,0,0 : show on
4930 repeat
4940 X=x mouse : Y=y mouse
4950 if hardkey=97 then hide : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back : show
4960 wait vbl : M=mouse key : until mouse key : hide on
4970 X1=X : Y1=Y : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : logic=back : wait 10
4980 if M=2 then logic=physic : wait 10 : cls : goto 200
4990 repeat
5000 opaque screen logic,0,0,SCR$(SC) : X1=x mouse : Y1=y mouse
5010 polyline X,Y to X1,Y1 to X,Y to X,Y
5020 screen swap : wait vbl
5030 until mouse key
5040 if mouse key=2 then logic=physic : wait 10 : cls : goto 200
5050 wait 10 : repeat
5060 opaque screen logic,0,0,SCR$(SC) : X2=x mouse : Y2=y mouse
5070 polyline X,Y to X1,Y1 to X2,Y2 to X,Y
5080 screen swap : wait vbl
```

```
5090 until mouse key

5100 if mouse key=2 then logic=physic : wait 10 : cls : goto 200

5110 wait 10 : logic=physic : fastcopy logic,back : show on : goto 4930
```

The first part of the routine draws the first angle of the triangle and the second part draws the rest of it.

And now, an eclipse type circle.

```
5120 rem CIRCLE

5130 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : show on : K=1

5140 repeat

5150 X=x mouse : Y=y mouse

5160 if hardkey=97 then hide : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back : show

5170 wait vbl : M=mouse key : until M

5180 if M=2 then hide on : wait 10 : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : cls : goto 200

5190 hide on : logic=back : wait 10

5200 SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200)

5210 repeat

5220 opaque screen logic,0,0,SCR$(SC) : X1=x mouse : Y1=y mouse

5230 earc X,Y,abs(X1-X),abs(Y1-Y),0,3600

5240 screen swap : wait vbl

5250 until mouse key

5260 if mouse key=2 then logic=physic : wait 10 : cls : goto 200

5270 logic=physic : fastcopy logic,back : show on : wait 10 : goto 5140
```

The ABS command stops the circle from drawing inwards.

No picture is complete without a bit of colour. This routine will fill an area of the screen with the current ink and pattern number, which can be altered with the SET PAINT and INK commands.

```
5280 rem FILL SHAPE

5290 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : show on : set paint 2,PSTYLE,1 : K=1

5300 repeat
```

```
5310 if mouse key=1 then hide : paint x mouse,y mouse : show

5320 if hardkey=97 then hide : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back : show

5330 wait vbl : until mouse key=2

5340 hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : wait 10 : cls : goto 200
```

Another interesting tool is the RAYS effect.

```
5350 rem RAYS

5360 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : K=1 : set line val(LTYPE$),L_SIZE,0,0 : show on : wait 10

5370 repeat

5380 X=x mouse : Y=y mouse : wait vbl

5390 M=mouse key : until M

5400 Y1=Y : X1=X : hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200)

5410 if M=2 then hide on : wait 10 : cls : goto 200

5420 logic=back : repeat

5430 X1=x mouse : Y1=y mouse

5440 opaque screen back,0,0,SCR$(SC) : polyline X,Y to X1,Y1

5450 if mouse key=2 then logic=physic : wait 10 : cls : goto 200

5460 screen swap : wait vbl

5470 until mouse key=1

5480 logic=physic : polyline X,Y to X1,Y1 : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : wait 2 : goto 5420
```

A K line is like a normal line except that it does not stop drawing itself when you reclick the mouse button. Here is the routine……

```
5490 rem K-LINE

5500 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : K=1 : set line val(LTYPE$),L_SIZE,0,0 : show on

5510 repeat

5520 X=x mouse : Y=y mouse : wait vbl

5530 M=mouse key : until M : wait 10

5540 Y1=Y : X1=X : hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200)

5550 if M=2 then goto 200
```

```
5560 logic=back : repeat

5570 opaque screen back,0,0,SCR$(SC) : polyline X,Y to X1,Y1

5580 X1=x mouse : Y1=y mouse

5590 if mouse key=2 then logic=physic : wait 10 : cls : goto 200

5600 screen swap : wait vbl

5610 until mouse key=1

5620 logic=physic : polyline X,Y to X1,Y1 : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : wait 2 : X=X1 : Y=Y1 : goto 5560
```

Two more interesting tools are the SPRAY and BRUSH.

```
6010 rem SPRAY

6020 off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : show on : K=1 : wait 10

6030 repeat

6040 K$=inkey$ : X=x mouse+4 : Y=y mouse+4 : wait vbl

6050 if hardkey=97 then hide : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : show

6060 if mouse key=0 then show on

6070 M=mouse key : until M : hide on : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : wait vbl

6080 if M=1 and X>=SIZE and X<=319-SIZE and Y>=SIZE and Y<=199-SIZE then plot X+rnd(SIZE),Y+rnd(SIZE) : plot X-rnd(SIZE),Y-rnd(SIZE) : plot X+rnd(SIZE),Y-rnd(S IZE) : plot X-rnd(SIZE),Y+rnd(SIZE) 6090 if SPEED>0 and M=1 then wait SPEED

6100 if M=2 then SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : wait 10 : cls : goto 200

6110 if mouse key=0 then show on

6120 goto 6030


6180 rem BRUSH

6190 hide on : off : cls : ink PE-1 : opaque screen back,0,0,SCR$(SC) : fastcopy back,logic : wait vbl : wait 10 : K=1 : show on

6200 repeat

6210 XM=x mouse : YM=y mouse

6220 if hardkey=97 then off : opaque screen logic,0,0,SCR$(SC) : fastcopy logic,back
```

```
6230 if M=1 and XM+BRUSH<320 and YM+BRUSH<200 then hide on : bar XM,YM to XM+BRUSH,YM+BRUSH else show on

6240 M=mouse key : until M=2

6250 if M=2 then hide on : wait 10 : SCR$(SC)="" : SCR$(SC)=screen$(logic,0,0 to 330,200) : cls : goto 200

6260 goto 6200
```

These are some of the more common tools used in an art package. There are more interesting features like MAGNIFY and RESIZE and all kinds of effects. But with these simple routines and a bit of imagination, there is no reason why you can't write your very own art package in STOS.

PROGRAMMING YOUR OWN SELECTION MENU

In some Atari demos they have a main menu where you can select a sub menu to load. We can do the same thing in STOS. This routine below allows you to move a character up and down a scrolling platform which is a kind of hallway with doors. You move your character to the door, press fire and the routine loads up the demo behind that door. For Example, going to door 12 will load up demo 12 and so on.

```
10 key off : hide on : curs off : flash off : mode 0

20 :

30 rem SET UP SCROLLING WINDOW SIZE AND VARIBLES

40 :

50 landscape 0,40,330,170,0,1 : MY=92 : Y=MY : XX=4 : XB=80 : YB=112 : IR=0 :
IL=5 : M=2 : IMR=0 : IML=0 : J=0 : JUMP=0 : S4=start(4) : S5=start(5) :
S6=start( 6) : dim DEMO$(12)

60 :

70 rem ENTER DEMO NAMES INTO ARRAY AND GET DOOR CO-ORDINATES

80 :

90 for D=1 to 12 : read DEMO$(D) : next D

100 A=block amount(S6,12) : dim DX(A+1),DY(A+1) : xy block
S6,varptr(DX(1)),varptr(DY(1)),12,A

110 :

120 rem GET PALETTE OF WORLD BLOCKS AND SET LOGIC SCREEN TO BACK
SCREEN

130 :

140 A=palt(S5) : logic=back

150 :

160 rem START OF LOOP

170 :

180 landscape logic,S5,S6,X,Y,0

190 :

200 rem CHECK TB (TOP OF SPRITE) AND BB (BOTTOM OF SPRITE)

210 :

220 TB=which block(S6,XB+8,Y+YB-40) : BB=which block(S6,XB+8,Y+YB-12)

230 :
```

```
240 rem CHECK IF SPRITE IS ON A WALKING BLOCK AND MOVE SCROLLING ABOUT

250 :

260 if TB=14 then JUMP=1

270 if BB<>14 and JUMP=1 and YB<112 then YB=YB+XX : goto 460

280 if BB<>14 and JUMP=1 then Y=Y+XX : goto 460

290 if BB=14 then JUMP=0

300 :

310 rem CHECK IF JOYSTICK PUSHED UP AND MOVE MAP OR SPRITE

320 :

330 if jup and Y>0 and TB<>14 and JUMP=0 then Y=Y-XX

340 if jup and Y=0 and TB<>14 and YB>30 and JUMP=0 then YB=YB-XX 350 :

360 rem CHECK IF SPRITE STILL IN AIR

370 :

380 J=joy : if BB<>14 and J=0 then JUMP=1

390 J=joy : if BB<>14 and J=4 then JUMP=1 400 J=joy : if BB<>14 and J=8 then JUMP=1 410 :

420 rem CHECK IF SPRITE OUTSIDE DOOR THEN GET THE DOOR NUMBER THEN PRINT NAME OF DEMO. IF FIRE PRESSED THEN GOTO LOADING DEMO ROUTINE 430 :

440 DB=which block(start(6),XB+8,Y+YB-24) : if DB=12 or DB=13 then gosub 690 else D00R=0

450 if D00R<>0 then locate 14,23 : print DEMO$(D00R) else locate 14,23 : print space$(20)

460 if fire and D00R<>0 then goto 750

470 :

480 rem CHECK FOR SPRITE MOVING LEFT AND ANIMATE IT

490 :

500 if jleft and XB>12 then XB=XB-XX : M=1 : inc IML : if IML=3 then inc IL : IML=0 : if IL=8 then IL=4 510 :

520 rem CHECK FOR SPRITE MOVING RIGHT AND ANIMATE IT

530 :

540 if jright and XB<284 then XB=XB+XX : M=2 : inc IMR : if IMR=3 then inc IR : IMR=0 : if IR=4 then IR=0 550 :
```

```
560 rem MAKE SURE CO-ORDINATES DON'T GO OVER THEIR LIMITS
570 :
580 if XB>284 then XB=284 : if XB<12 then XB=12
590 if Y>MY then Y=MY
600 :
610 rem PUT BOBS IN SCREEN FACING EITHER LEFT (M=1) OR RIGHT (M=2)
620 :
630 if M=1 then bob logic,S4,IL,XB,YB,0
640 if M=2 then bob logic,S4,IR,XB,YB,0
650 :
660 rem SWAP SCREEN AND GO BACK TO START OF LOOP
670 :
680 screen swap : wait vbl goto 180
690 :
700 rem THE DOOR CHECKING ROUTINE
710 :
720 for D=1 to 12
730 if XB>=DX(D) and XB<=DX(D)+32 and Y+YB>DY(D) then D00R=D
740 next D : return
750 :
760 rem THE DEMO LOADING ROUTINE
770 :
780 logic=physic : cls : F$="LOADING"+str$(D00R)+".PRG" : locate 0,7 : print F$
790 :
800 rem THE DEMO NAMES
810 :
820 data "DEMO ONE","DEMO TWO","DEMO THREE","DEMO FOUR","DEMO
FIVE","DEMO SIX","DEMO SEVEN","DEMO EIGHT","DEMO NINE","DEMO TEN","DEMO
ELEVEN","DEMO TWELVE"
```

The graphics are wall blocks that make up the hallway and there are some other blocks that make up the doors. There are four 16x16 blocks that make up the door and all we

do is check for them colliding with the main character. A number is returned and we use that to discover which door number he is at.

SPRITES QUESTIONS AND ANSWERS

STOS sprites can certainly be a pain sometimes. This article aims to make using them a little easier to understand.

The default sprites in STOS are software sprites, meaning that STOS has to do a lot of work with them which makes them slow and flicker a lot. It spends a fair bit of processor time updating them, and with them working on interrupt also adds to problems.

Here are a few typical questions about them along with answers…

QUESTION: The command 'sprite 16,100,120,4' produces an error.

ANSWER: This is because STOS only allows up to 15 sprites on the screen at the same time. A higher value will produce errors.

QUESTION: The sprite flickers when it moves.

ANSWER: This is because of the monitor picture tube updating causing the sprite to flicker when it passes it. Use the 'screen swap' command to avoid this.

QUESTION: Can I get the sprites moving more smoothly?

ANSWER: Yes, STOS updates the sprites every 50th of a second. This can lead to problems with speed and movement so you're best of using the 'update off' command and updating them yourself with the update command like this….

```
10 key off : hide : update off

20 sprite 1,X,Y,4 : update
```

QUESTION: The more sprites I use, the slower they get.

ANSWER: Again this is due to STOS using a lot of time to update them all, try using fewer sprites on the screen at the same time. For a shoot em up game, you could have about five sprites on screen at any one time and replace each one as it gets killed.

QUESTION: Large sprites are difficult to handle.

ANSWER: Yes they are…the bigger the sprite, the more time STOS uses to update them. If you must have large sprites then use as few as possible. About two or three on-screen at the same time.

QUESTION: The sprite flashes on the screen.

ANSWER: This is because it has colour 2 in it, STOS always flashes this colour. Use the "flash off" command.

QUESTION: If I place the sprite on a picture, the colours of the sprites are different to what they were.

ANSWER: Pictures and sprites have their own separate palettes. When you load a picture to the screen or unpack one from a bank STOS adjusts its palette to one of the screens so the sprite gets the same palette. Load your sprites back into the 'Sprite

Definer', grab the palette from your background picture and re-colour the sprites with the colours of the pictures.

QUESTION:If I place a sprite on the screen, its colours change.

ANSWER: Again this is due to the sprite and the screen having two different palettes. Unlike pictures STOS doesn't adjust his palette to one of the sprites, you have to do it yourself like this…

```
10 key off : mode 0 : flash off
20 XP=hunt(start(1) to start(1)+length(1),"PALT")
30 XP=XP+4 : for I=0 to 15 : colour I,deek(I*2+XP) : next I
```

QUESTION: The sprite hasn't appeared on the screen.

ANSWER: There are two ways this can happen.

1. The image number you used is a blank space in the sprite

bank, so STOS displays a blank space.

2. You've given the X and Y parameters a negative value, IE: you've

typed 'sprite 1,-50,-100,4', putting it off the screen.

QUESTION: Can I display sprites in medium resolution?

ANSWER: Yes, define them with the 'Sprite2' accessory.

QUESTION: How can I animate about 20 images, ANIM won't do it.

ANSWER: The "anim" command only stores about 10 to 15 images at a time. You'll have to animate them yourself like this.

```
10 key off : flash off : mode 0
20 for X=1 to 20
30 sprite 1,100,100,X : wait 20
40 next X
```

The 'wait' command controls the speed of the animation.

QUESTION: How do I control a sprite using the joystick.

ANSWER: Don't use the move command for this, it doesn't stop when you want it to. Instead, use this routine…

```
10 key off : mode 0
20 if jleft then X=X-2
30 if jright then X=X+2
40 if jup then Y=Y-2
```

```
50 if jdown then Y=Y+2

60 sprite 1,X,Y,1 : wait vbl

70 goto 20
```

QUESTION: Why can’t I move the sprite against a scrolling background?

ANSWER: Because STOS updates scrolling quicker than it updates sprites causing them to jerk. The best thing to do is use the “bob” and ‘world’ commands from the Missing Link extension.

Well, that’s it for this article. I hope it’s been useful. Most STOS coders use pre-shifted sprites these days but if you are just learning then I’m sure you’ll find this article useful.

STACKING A MEMORY BANK

As you know, there are only fifteen memory banks that you can use in STOS. Yet some games have more than fifteen pictures or music files overcoming the fifteen-bank limit. This is done by simply sticking all the files into one bank on top of each other. This is called Stacking a Bank.

Let's say we have five pieces of chip music that we want to save along with the program in one bank and call each piece when we need it. Well first to get them all in the bank we need to add up the total length of the files and reserve a bank to this length. If you put your five selected tunes on a disk then type 'dir' you should get a list like this.

Drive A:

MADMAX.MUS 4200

KILLING.MUS 2100

CIRCUS.MUS 8244

ALEC.MUS 4774

STOMP.MUS 8000

Here we have first the filenames followed by the file length, the file lengths need to be added up and then a bank reserved…..

4200+2100+8244+4774+8000=27318

10 reserve as work 5,27320+100

Note how the value in line 10 is slightly higher than the calculated figure. This is because we must always reserve a round figure for the length, we then add 100 bytes just to make a bit more space in the bank. Now we can add more lines to this routine to load the music files into the bank.

20 bload "MADMAX.MUS",start(5)

30 bload "KILLING.MUS",start(5)+4200

40 bload "CIRCUS.MUS",start(5)+4200+2100

50 bload "ALEC.MUS",start(5)+4200+2100+8250

60 bload "STOMP.MUS",start(5)+4200+2100+8250+4780

70 rem SAVE BANK

80 bsave "M_BANK.DAT",start(5) to start(5)+length(5)

The first file has loaded into the large bank starting from the start of the bank so it has taken up 4200 bytes of the bank which is the length of the first file (MADMAX.MUS). To load the second file in we have to make sure it slots in after the first one so we load it into the bank starting at the position where the first file ends. The next file has to load in and slot in the bank after the first two so we add the values of the first two files to give us the starting position where to load the third file.

So basically, we are loading in each file after the other adding up the values of the previous files to find the start position of the present file in the large reserved bank. Rather than have all this value+value bit we can add up the values into one length like so.

```
20 bload "MADMAX.MUS",start(5)

30 bload "KILLING.MUS",start(5)+4200 : rem Length of first file

40 bload "CIRCUS.MUS",start(5)+6300 : rem Length of files 1 and 2

50 bload "ALEC.MUS",start(5)+14550 : rem Length of files 1,2 and 3

60 bload "STOMP.MUS",start(5)+19330 : rem Length of files 1,2,3 and 4
```

Note it's also important that file lengths that are not even must be rounded up to the nearest even figure for the bank to stack properly.

To use the stacked bank in our program we just reserve a bank to the full length and bload it in. Now the easiest and quickest way to play the music is like this. First, we need two arrays that will hold the start address of each file, and the length of each file. The music can easily be played by one line, saving a load of IF statements.

```
10 reserve as work 5,27320+100

20 bload "M_BANK.DAT",5

30 dim MUS_ST(5) : rem Reserve array for start addresses

40 dim MUS_LE(5): rem Reserve array for file lengths

50 MUS_ST(1)=0 : MUS_ST(2)=4200 : MUS_ST(3)=6300 : MUS_ST(4)=14550 :

MUS_ST(5)=19330

60 MUS_LE(1)=4200 : MUS_LE(2)=2100 : MUS_LE(3)=8250 : MUS_LE(4)=4780 :

MUS_LE(5)=8000

70 input "Choose tune to play (1 to 5)";PL

80 if PL=0 or PL>5 then goto 70

90 rem Play chosen music file

100 A=musauto(start(5)+MUS_ST(PL),1,MUS_LE(PL))

110 wait key

120 A=musauto(0,0,0) : goto 70
```

You can use the same method with packed pictures. In your game you just load in the bank of stacked binary packed pictures and call them up in the same way……for example.

```
10 reserve as work 10,50000

20 bload "PICS.PAC",10
```

```
30 dim SCR_ST(4)

40 SCR_ST(1)=4000 : SCR_ST(2)=3200 : SCR_ST(3)=6500 : SCR_ST(4)=9000

50 for X=1 to 5

60 unpack start(5)+SCR_ST(X) : wait 50

70 next X
```

So, there you have it. With this method, you can have as many binary files in a bank as memory permits, and put an end to the bank limit.

STOS ADDRESS BOOK

Welcome to this tutorial on writing a database program in STOS. A simple database that will store information in the way of names, addresses, and other details. I will call this program "STOS Address Book". Note there are no prizes for guessing a better name.

First, what is a database? Well, it's a program that stores information entered by the user, and allows them to view it later. But why write a database to store names and addresses when you can quite simply use an address book to write in? Well, there are certain advantages…..for example, a program can find the info quicker than someone searching through a book. Also, should you wish to delete someone's entry at any time then you can do it quite easily without having blotted out entries.

Each entry is entered into a field. In other words, an element of an array. So, let's first set the screen up.

```
5 rem —ADDRESS BOOK

10 key off : curs off : hide on : flash off : mode 1

15 rem—SET UP ARRAYS FOR FILE INFO

20 dim SURNAME$(1000) : dim NAME$(1000) : dim ADDR$(1000) :

dim T0WN$(1000) : dim POST$(1000) : dim PHONE$(1000) :

dim INFO$(1000)

25 rem—SET UP VARIABLES

30 FILE=0 : FILES=0
```

Most databases are displayed in the medium resolution which makes them look more professional. There are five fields: Surname, Names, Addresses, Towns, Postcode, Phone, and Info. Note the way the TOWN array is spelled. The O is a nought. This stops STOS from interpreting the word as to WN$

Each field has space reserved for 1000 entries. The more entries we have, the more memory is used. Now, let's set up the rest of the screen.

```
35 rem—OPEN FIRST WINDOW AND ENTER TITLE"

40 windopen 1,0,0,80,3,3,1

50 cdown : centre "ADDRESS BOOK"

55 rem—OPEN SECOND WINDOW AND ENTER OPTIONS"

60 windopen 2,59,3,21,17,3,1

70 locate 1,2 : print "(C)REATE A FILE"

80 locate 1,4 : print "(D)ELETE A FILE"

90 locate 1,6 : print "(F)IND A FILE"

100 locate 1,8 : print "(L)IST ALL FILES"
```

```
110 locate 1,10 : print "(P)RINT A FILE"

120 locate 1,12 : print "(Q)UIT AND SAVE"

125 rem—OPEN THIRD WINDOW FOR PROGRAM INFO

130 windopen 3,0,20,80,3,3,1

140 centre "CHOOSE AN OPTION"
```

This part of the program opens three windows which displays the title, program options, and the bottom window is used for program information, meaning that we use it prompting the user what to do.

The next thing to do is to use the blank part of the screen so we can print the field names but first, we have to leave the windows.

```
145 rem—LEAVE WINDOW AND GOTO SCREEN

150 qwindow 0
```

This is an undocumented feature of STOS. The manual tells us that when using the window commands the values range from 1 to 15 while nought is the STOS system window. You can however leave the windows completely without going to another or deleting them by using a "qwindow 0". Now we can print the field names on-screen without affecting the windows.

```
155 rem—PRINT FIELD NAMES

160 locate 0,4 : print "SURNAME:"

170 locate 0,6 : print "NAME:"

180 locate 0,8 : print "ADDRESS:"

190 locate 0,10 : print "TOWNS:"

200 locate 0,12 : print "POSTCODE:"

210 locate 0,14 : print "PHONE:"

220 locate 0,16 : print "INFO:"

225 locate 0,18 : print "FILES:";FILES
```

Everything's set up. Now we just have to wait for the user to select an option.

```
230 rem—WAIT FOR AN INPUT

240 K$=input$(1)

250 rem—CHECK WHICH KEY HAS BEEN PRESSED

260 if K$="C" or K$="c" then goto 400

270 if K$="D" or K$="d" then goto 500

280 if K$="F" or K$="f" then goto 600
```

```
290 if K$=”L” or K$=”l” then goto 700

300 if K$=”P” or K$=”p” then goto 800

310 if K$=”Q” or K$=”q” then goto 900

320 K$=”” : goto 240
```

The variable K$ is used to store the user's selection. At the moment he only chose C to create a file…the program will then go to line 400 which is the input routine. Line 320 clears the variable and returns to line 240 ready to await another selection.

```
400 rem—-CREATE A FILE

410 curs on : inc FILES

420 rem—GET SURNAME

430 X=8 : Y=4 : gosub 2000 : SURNAME$(FILES)=K$

440 rem—GET NAME

450 X=5 : Y=6 : gosub 2000 : NAME$(FILES)=K$

460 rem—GET ADDRESS

470 X=8 : Y=8 : gosub 2000 : ADDR$(FILES)=K$

480 rem—GET TOWNS

490 X=6 : Y=10 : gosub 2000 : T0WN$(FILES)=K$

500 rem—GET POSTCODE

510 X=9 : Y=12 : gosub 2000 : POST$(FILES)=K$

520 rem—GET PHONE NUMBER

530 X=6 : Y=14 : gosub 2000 : PHONE$(FILES)=K$

540 rem—GET INFO

550 X=5 : Y=16 : gosub 2000 : INFO$(FILES)=K$

555 rem—FILE CREATED SO GO BACK AND WAIT FOR NEXT INPUT

560 K$=”” : curs off : gosub 2070 : goto 225
```

This is a kind of setup to enter the information needed. The X and Y variables hold the X and Y coordinates of the text cursor. Notice at the start of this part we turn the cursor on and add one to the FILES variable. This is the number of the file we are currently creating.

The X and Y variables are needed to position the text cursor just after each field name…IE SURNAME, NAME, etc… The routine then does a subroutine to the routine that enters the information into a variable. Once the file has been created the program turns off the cursor, does a subroutine to clear the wording entered off the screen then goes back to wait for another selection from the user.

```
2000 rem—INPUT ROUTINE
2010 locate X,Y
2020 input “”;K$
2030 if K$=”” then K$=”UNKNOWN”
2040 return
```

The actual input routine. Note how we use the locate command to position the cursor at the right place for every field name. Line 2020 allows you to type in the information while line 2030 checks if the users just press the enter key without typing anything in.

```
2070 rem—CLEAR INFO OFF SCREEN
2080 wait 30
2090 locate 8,4 : print space$(50)
2100 locate 5,6 : print space$(53)
2110 locate 8,8 : print space$(50)
2120 locate 6,10 : print space$(52)
2130 locate 9,12 : print space$(49)
2140 locate 6,14 : print space$(52)
2150 locate 5,16 : print space$(53)
2160 return
```

The final part clears the wording off the screen once the file is created. The space$ command just prints a line of spaces to clear it.

Now we are adding a Delete File function to our Address Book program. Here is the main routine for it.

```
600 rem—DELETE A FILE
610 if FILES=0 then goto 310
620 locate 5,6 : curs on : input “”;N$ : curs off : if N$=”” then goto 310
630 rem—LOOK FOR FIRST THREE LETTERS OF N$ IN NAME$ ARRAY
635 FILE=0
640 FOUND=0 : repeat
650 inc FILE : if left$(NAME$(FILE),3)=left$(N$,3) then FOUND=1 : goto 690
660 until FILE=FILES
670 rem—IF THE FILE IS’NT FOUND THEN GO BACK TO INPUT
```

```
680 if FOUND=0 then qwindow 3 : clw : curs off : centre "FILE NOT FOUND" : wait 100
: qwindow 0 : locate 5,6 : print space$(53) : qwindow 3 : clw : goto 200

690 rem—FILE FOUND SO PRINT INFO

700 locate 8,4 : print SURNAME$(FILE)

710 locate 5,6 : print NAME$(FILE)

720 locate 8,8 : print ADDR$(FILE)

730 locate 6,10 : print T0WN$(FILE)

740 locate 9,12 : print POST$(FILE)

750 locate 6,14 : print PHONE$(FILE)

760 locate 5,16 : print INFO$(FILE)

770 locate 0,18 : print "FILES:";FILES

780 rem—ASK USER IF HE WANTS TO DELETE THE FILE

790 qwindow 3 : clw : curs off : centre "DELETE THIS FILE (Y/N)?" : D$=input$(1)

800 if D$="n" or D$="N" then qwindow 0 : gosub 3000 : goto 640

810 rem—DELETE FILE AND SORT THEM INTO ORDER

820 for X=FILE to FILES

830 SURNAME$(X)=SURNAME$(X+1)

840 NAME$(X)=NAME$(X+1)

850 ADDR$(X)=ADDR$(X+1)

860 T0WN$(X)=T0WN$(X+1)

870 POST$(X)=POST$(X+1)

880 PHONE$(X)=PHONE$(X+1)

890 INFO$(X)=INFO$(X+1)

900 next X : dec FILES

910 rem—GO BACK TO OPTIONS

920 qwindow 0 : gosub 3000 : qwindow 3 : clw : goto 200
```

What this routine does is it first checks if there are any files in memory and if so asks the user for the first name of the person they wish to delete. The routine finds the name then asks the user if he wants to delete it.

Note that only the first three letters of the name are searched for, this increases speed when searching. If the user doesn't wish to delete the first file, then the program will continue to list the other files under that name until either it reaches the end of the user deletes a file.

The program also uses window three to display the information for the function. It also does a gosub to a clearing routine to get rid of the excess wording on screen. It then deletes the selected record by doing a loop from FILE which is the file number to be deleted, to FILES which is the number of records held in memory. After this it subtracts the FILE variable by one then goes back to the selection routine. As a file has been removed the program brings all the other records back to replace the deleted file.

I have added a FIND FILE function to the database program. This allows us to enter the name of the person whose details we wish to find and the program will list every name it finds under that input, allowing us to step through each one till we find the one we want.

```
1000 rem—FIND A FILE

1010 if FILES=0 then goto 310

1020 locate 5,6 : curs on : input "";N$ : curs off : if N$="" then goto 310

1030 rem—LOOK FOR FIRST THREE LETTERS OF N$ IN NAME$ ARRAY

1040 FILE=0

1050 FOUND=0 : repeat

1060 inc FILE : if left$(NAME$(FILE),3)=left$(N$,3) then FOUND=1 : goto 1100

1070 until FILE=FILES

1080 rem—IF THE FILE ISN'T FOUND THEN GO BACK TO INPUT

1090 if FOUND=0 then qwindow 3 : clw : curs off : centre "FILE NOT FOUND" : wait 100 : qwindow 0 : locate 5,6 : print space$(53) : qwindow 3 : clw : goto 200

1100 rem—FILE FOUND SO PRINT INFO

1110 locate 8,4 : print SURNAME$(FILE)

1120 locate 5,6 : print NAME$(FILE)

1130 locate 8,8 : print ADDR$(FILE)

1140 locate 6,10 : print T0WN$(FILE)

1150 locate 9,12 : print POST$(FILE)

1160 locate 6,14 : print PHONE$(FILE)

1170 locate 5,16 : print INFO$(FILE)

1180 locate 0,18 : print "FILES:";FILES

1190 rem—ASK USER IF HE WANTS TO CONTINUE THE LISTING

1200 qwindow 3 : clw : curs off : centre "FILE NO:"+str$(FILE)+" NEXT FILE (Y/N)" : D$=input$(1)

1210 if FILE=FILES then qwindow 0 : gosub 3000 : goto 3100
```

1220 if D$=”Y” or D$=”y” and FILE<FILES then qwindow 0 : gosub 3000 : goto 1050

This is almost the same as the Delete file function only it allows you to step through the records stored so you can read them. Notice how the program only checks the first three letters of the name for speed. The final part of the program checks if the user wants to continue and only goes back to list another record if the variable FILE (no of present file) is lower than the variable FILES (total no of files stored). Now let’s look at listing all the files.

1230 rem —LIST ALL FILES

1290 FILE=0 : if FILES=0 then goto 310

1300 for FILE=1 to FILES

1310 locate 8,4 : print SURNAME$(FILE)

1320 locate 5,6 : print NAME$(FILE)

1330 locate 8,8 : print ADDR$(FILE)

1340 locate 6,10 : print T0WN$(FILE)

1350 locate 9,12 : print POST$(FILE)

1360 locate 6,14 : print PHONE$(FILE)

1370 locate 5,16 : print INFO$(FILE)

1380 locate 0,18 : print “FILES:”;FILES

1390 rem—ASK USER IF HE WANTS TO CONTINUE THE LISTING

1400 qwindow 3 : clw : curs off : centre “FILE NO:”+str$(FILE)+”

NEXT FILE (Y/N)” : D$=input$(1)

1410 if FILE=FILES then qwindow 0 : gosub 3000

1420 if D$=”Y” or D$=”y” and FILE<FILES then qwindow 0 : gosub 3000

else qwindow 0 : gosub 3000 : goto 3100

1430 next FILE

This routine uses a loop to step through each file in turn and print it on the screen. The user is then asked if he would like to continue listing or to terminate. If he continues then the loop continues. If the variable FILE equals the same as FILES (number of files stored) then the program terminates itself.

If you want to see this working then I have supplied the complete STOS Code rather than type out this listing. You can use the routines for other programs if you so wish.

STOS BASIC ROUTINES

A collection of STOS Basic routines I programmed. Taken from Stosser, Power and ST+ diskzines

HIGH SCORE TABLE:

```
10 key off : cls
20 dim NAME$(10) : dim SC(10)
30 for X=1 to 10 : read NAME$(X) : next X
40 for X=1 to 10 : read SC(X) : next X
50 for X=1 to 10
60 print NAME$(X);" ";SC(X)
70 next X
80 print : input "Name:";A$ : input "Score:";B
85 for X=1 to 10
90 if B>SC(X) then swap NAME$(X),A$ : swap SC(X),B
100 next X : cls : goto 50
150 data "DEANO","MICK","JEFF","MIKE","STEVE","TOM","PHIL","PETE","JACK","ANDY"
160 data 1000,900,800,700,600,500,400,300,200,100
```

POKING CHARACTERS INTO A BANK:

```
10 reserve as work 5,400 : S5=start(5)
20 input "STRING:";A$ : LE=len(A$)
30 for X=1 to LE : poke S5+X,asc(mid$(A$,X,1)) : next X
40 for X=1 to LE : print chr$(peek(S5+X)); : wait 5 : next X
```

RANDOM NUMBERS IN AN ARRAY WITH NONE REPEATING:

```
10 cls
20 dim NUMB(12)
30 for LOOP=1 to 12
40 NUMB(LOOP)=0
50 next LOOP
60 for LOOP=1 to 12
70 repeat
```

```
80 EL=rnd(11)+1
90 until NUMB(EL)=0
100 NUMB(EL)=LOOP
110 next LOOP
120 for X=1 to 12 : print NUMB(X); : next X
```

RANDOM WORDS IN AN ARRAY WITH NONE REPEATING:

```
1 rem WRD$() holds the words to be placed in a random order
2 rem RWRD$() hold the same words but in a random sequence
100 key off
110 dim WRD$(12),RWRD$(12) : for W=1 to 12 : read WRD$(W) : next W
120 rem clear array
130 for W=1 to 12 : RWRD$(W)="" : next W
140 for W=1 to 12
150 R=rnd(11)+1 : if RWRD$(R)<>"" then 150
160 RWRD$(R)=WRD$(W)
170 next W
180 for W=1 to 12
190 print using "##";W;"  ";RWRD$(W)
200 next W
210 rem The 12 words, put anything here!
220 data "One","Two","Three","Four","Five","Six","Seven","Eight","Nine","Ten","Eleven","Twelve"
```

SIN WAVE LINES:

```
10 curs off : hide on : key off : mode 0
20 palette $0,$11,$577,$377,$177,$375,$573,$771,$750,$730,$700,$702,$703,$705,$727,$747
30 def scroll 1,64,16 to 256,200,0,8
40 logic=back : A=0 : C=1
50 X1=160+sin(A/21.6)*95
60 X2=160+sin(A/16.6)*95
```

```
70 Y1=58+cos(A/22.6)*40
80 Y2=58+cos(A/12.6)*40
90 ink C : C=(C mod 14)+2
100 draw X1,Y1 to X2,Y2
110 scroll 1 : screen swap
120 wait vbl : inc A
130 goto 50
```

SIN WAVE BOXES:

```
10 curs off : hide on : key off : mode 0
20 palette $0,$11,$577,$377,$177,$375,$573,$771,$750,$730,$700,$702,$703,$705,$727,$747
30 def scroll 1,64,16 to 256,200,0,8
40 logic=back : A=0 : C=1 : A=musauto(start(5),1,7858)
50 X1=160+sin(A/21.6)*95
60 X2=160+sin(A/16.6)*95
70 Y1=58+cos(A/22.6)*40
80 Y2=58+cos(A/12.6)*40
90 ink C : C=(C mod 14)+2
100 bar X1,Y1 to X2,Y2
110 scroll 1 : screen swap
120 wait vbl : inc A
125 if inkey$=" " then A=musauto(0,0,0) : stop
130 goto 50
```

FILL UP THE SCREEN WITH TILES

```
10 key off : hide on : curs off : flash off : mode 0
20 rem RESERVE BANK AS SCREEN AND ASSIGN LOGIC TO IT
30 reserve as screen 7 : logic=7
40 rem PLACE SPRITE TILE ON SCREEN AND MAKE A COPY OF IT
50 sprite 1,0,0,1 : put sprite 1 : wait vbl : sprite off
60 rem COPY THE TILE BLOCK TO A variable
```

```
70 T$=screen$(logic,0,0 to 16,16)
80 rem START OF LOOP
90 for X=0 to 330 step 16 : for Y=0 to 199 step 16
100 rem PLACE COPYS OF TILE AT CO-ORDINATES X AND Y
110 screen$(logic,X,Y)=T$
120 rem END LOOP
130 next Y : next X
140 rem SET SCREEN BACK TO NORMAL AND SHOW THE TILE PICTURE
150 logic=physic : screen copy 7 to physic
```

This routine fills the screen with copies of a sprite that has been copied on screen so it can be grabbed into a variable with SCREEN$. The sprite has to be a size of 16×16 pixels so the loop can make copies of it fit evenly onto the screen. The size could be changed but it must be in sizes of 16 so you could copy a 32×32 sprite block if you wish. This makes sure the loop can fit into the screen perfectly. Note the step size is that of the sprite size. I've also got it to build it up in a memory bank so you don't see it build up on the screen. This is quite fast but you could change it to grab a tile block from a screen in memory instead of the copied sprite block.

SIMPLE TEXT SCROLLER

```
10 rem SCROLLING TEXT LINE
20 :
30 rem SET SCREEN
40 key off : mode 0 : curs off : hide on
50 :
60 rem DEFINE SCROLLING AREA
70 Y=ygraphic(10) : Y1=ygraphic(11)
80 def scroll 1,0,Y to 320,Y1,-4,0
90 :
100 rem DEFINE TEXT TO SCROLL
110 T$="This is a simple scrolling routine which scrolls text in a variable across the screen……. "
120 :
130 rem SCROLL THE TEXT
```

```
140 for L=1 to len(T$)

150 locate 39,10

160 print mid$(T$,L,1)

170 wait vbl : scroll 1

180 wait vbl : scroll 1

190 next L

200 rem START AGAIN

210 goto 130
```

It’s more or less straight forward. The use of the ygraphic command converts the text coordinates to graphic coordinates so the def scroll command knows where to place the scrolling zone. The above example will place the text at coordinates at 39,10 so the variable Y sets the top part of the scrolling zone downwards while Y1 sets the bottom part.

```
Y———————————————————————————-

THE SCROLLING ZONE FOR THE TEXT

Y1———————————————————————————
```

So Y equals y co-ordinate 10 whilst y1 equals the next line down which is 11. So the thing to remember is Y equals the y coordinate where you wish to position the text and Y1 equals the next text line. Try changing the values to see how it works. The loop does the scrolling. It works by placing one character at a time at the edge of the screen then scrolling it along by eight pixels then printing the second character at the same coordinate and moving that along. It continues this until the contents of the full string (T$) has been printed and scrolled. It's not very easy to explain so the best way to learn is to play around with the routine.

GET COLOUR HEX VALUES FROM A PICTURE IN MEMORY BANK

```
10 key off : curs off : hide : flash off : mode 0

20 get palette (5)

30 for X=0 to 15 : print hex$(colour(X)) : next X
```

CHANGING PALETTE VALUES OF PICTURES DRAWN WITH STOS

```
10 key off : hide : flash off : curs off : mode 0

20 for X=0 to 15 step 10 : paper x : bar X,10 to X+10 : next X

30 rem Coloured boxes on-screen, now lets change colours
```

40 wait key

50 palette $0,$777,$756,$676….etc: Rem put 16 hex numbers on a line and

change one or two

60 wait key

70 colour 1,$0 : rem change colour one to black

This method won't work with a picture saved from an art package. What I think you need to do is poke the new hex value into the screen bank. The first 32 bytes of the bank is the screen palette.

UNPACK AN MBK SCREEN

STOS has two commands for packing and unpacking pictures. These are PACK and UNPACK. The MBK file has been packed with the STOS compact accessory so you need to use the unpack command in your routine to unpack it like so.

10 key off : hide : curs off : flash off : mode 0

20 load"picture.mbk",5

30 unpack 5,back

40 screen copy back to physic

So, in other words, line 20 loads the packed picture into bank five. For MBK files you don't usually need to reserve a bank first. The unpack command then unpacks the picture to the background screen then the whole screen is copied to the main physic screen. Note you can also unpack a screen into a bank.

NICE FADE EFFECT

10 key off : hide on : flash off : curs off : mode 0

20 rem first set all colours to white

30 wait 10

40 fade 5,$777,$777,$777,$777,$777,$777,$777,$777,$777,$777,$777,$777,$777,

$777,$777,$777,$777

50 rem wait for fade to happen

60 wait 80

70 rem now fade colours back to the present palette

80 fade 5,$523,$777,$0,$123,$232,…etc

FIND WHICH KEYS ARE ASCII CODES AND WHICH ARE SCANCODES

10 key off : hide on: curs off

20 print"press a key"

```
30 c$=inkey
40 if c$="" then goto 30
50 if not(scancode) then print"This is an ascii character" else the scancode
for this key is ";scancode
60 goto 20
```

PRINT CONTENTS OF A FOLDER

```
10 key off : hide on
20 dir$="DATA": rem name of the folder
30 print all files to the screen
40 P$=".": N$=dir first$(P$-1): if N$="" then end
50 print N$
60 repeat
70 N$=dir next$
80 print N$
90 until N$=""
100 dir$="A:": rem close folder and default back to root directory
110 rem print contents of folder on printer
120 dir$="DATA"
130 ldir
140 dir$="A:"
```

GET RID OF THE MISSING LINK REGISTRATION REMINDER MESSAGE

In the full version of the missing link you get a third extension (EXS) which contains a command "mostly harmless" to gets rid of that message. Type out this routine.

```
10 A=mostly harmless (1,2,3,4,5)
20 put key "NEW"
```

Save this in the root directory (not in a folder) of your STOS disk under the name of 'autoexec.bas'. Now when you load STOS this file will run, a message will appear saying 'You got it' and the NEW command pops up. Press return and the message has gone.

PASSWORD ROUTINE THAT STORES IN A VARIABLE

```
10 key off : hide on : curs off : mode 0
20 locate 0,10 : centre"Please enter the password."
30 A$=input$(5) : rem password is five characters long
```

40 if A$=”POWER” or A$=”power” then locate 0,12:centre “Okay, loading.” else locate 0,12: centre”That is not the password, try again.” : goto 30

SCREEN UNPACKING TIP

Normally when you unpack a screen from a bank, the screen palette changes. The method to fix it is to use the missing link “floodpal” command like this:

10 key off : flash off : curs off: mode 0

20 reserve as screen 5 : unpack 4,5 : floodpal 0 : wait vbl

30 get palette (5) : screen copy 5 to back : screen copy back to logic

The wait vbl is important in this routine.

Thanks to Tony Greenwood for this tip.

HAVE SOMETHING HAPPEN AFTER TWO HOURS

10 time$=”10:00:00?

20 repeat

30 until time$=”12:00:00?

40 print “Two hours in real-time have passed.”

SIMPLE SPELL CHECKER

How this routine works are to hold a list of words in arrays and check each word of the document against the ones in the arrays. In the spell checking part of a WP, all words are stored in alphabetical order, so first we can store a list of words beginning with A in a data statement. We can then use the SORT command to put them in order then the MATCH command to check each word in the document against the words in the array. Assuming each word of the document is held in an array then this routine does the checking.

10 rem A$ array holds all words in doc, B$ array holds words to check

against

20 dim B$(3)

30 for X=1 to 10 : read B$(X) : next X

40 sort B$

50 rem Check all words in doc against words in B$

60 inc W : W$=A$(W) : POS=match(B$(0),W$)

70 if POS<0 then goto 90

80 if W<10 then goto 60

90 rem WORD NOT FOUND

```
100 for X=0 to 2 : print B$(X) : next X

110 data”love”,”lovely”,”lover”
```

What happens here, is that the SORT command puts the words you want to check against the doc in alphabetical order, then the W$ variable gets each word of the document and checks it against the sorted word array (B$). If POS is less than nought then the routine has found a word it doesn’t understand and will print out the list of words it does know. What you can do for speed is just list the words beginning with the first letter of the word that the routine couldn’t find. Like this.

```
120 rem A$ holds all words beginning with A

130 if left$(W$,1)=”a” then goto 140

140 for X=0 to 2 : print B$(X) : next X
```

RPG AND ADVENTURE GAME EXITS ROUTINES

The quickest way of reading things like exits and such in a maze of rooms is to store all the info in arrays, then check which room the player is in and set zones in that room.

```
10 key off : curs off : mode 0

20 dim MAP(5,4) : dim XZ1(5,4),YZ1(5,4),XZ2(5,4),YZ2(5,4) : ROOM=1

25 rem SET UP MAP EXIT VALUES

30 for X=1 to 5 : for Y=1 to 4

40 read MAP(X,Y)

50 next Y : next X

52 rem SET UP ZONE CO-ORDINATES FOR EACH ROOM

55 for X=1 to 5 : for Y=1 to 4

66 read A,B,C,D

70 XZ1(X,Y)=A : YZ1(X,Y)=B : XZ2(X,Y)=C : YZ2(X,Y)=D

75 next Y : next X

76 rem DRAW EXIT BOXES AND INFO OF WHERE EACH ONE GOES

80 home : print “ROOM”;ROOM : locate 0,6

90 for X=1 to 4

100 box XZ1(ROOM,X),YZ1(ROOM,X) to XZ2(ROOM,X),YZ2(ROOM,X)

110 set zone X,XZ1(ROOM,X), YZ1(ROOM,X) to XZ2(ROOM,X),  YZ2(ROOM,X)

115 print “EXIT”;X;” GOES TO ROOM”;MAP(ROOM,X)

120 next X

125 rem WAIT FOR PLAYER TO CHOOSE A ZONE
```

```
130 repeat
140 EXIT=zone(0)
150 until EXIT<>0 and mouse key=1
155 rem SET ROOM VALUE TO NEW ROOM
160 ROOM=MAP(ROOM,EXIT) : goto 80
1000 rem EXIT VALUES FOR EACH OF THE FIVE ROOMS IN MAZE
1010 data 2,3,4,5
1020 data 1,2,3,4
1030 data 2,1,3,4
1040 data 3,2,1,5
1050 data 1,2,3,4
2000 rem ZONE CO-ORDINATES FOR ROOM ONE
2010 data 10,20,20,40
2020 data 30,20,40,40
2030 data 50,20,60,40
2040 data 70,20,80,40
2060 rem ZONE CO-ORDINATES FOR ROOM TWO
2070 data 10,20,20,40
2080 data 30,20,40,40
2090 data 50,20,60,40
2100 data 70,20,80,40
2120 rem ZONE CO-ORDINATES FOR ROOM THREE
2130 data 10,20,20,40
2140 data 30,20,40,40
2150 data 50,20,60,40
2160 data 70,20,80,40
2170 rem ZONE CO-ORDINATES FOR ROOM FOUR
2180 data 10,20,20,40
2190 data 30,20,40,40
2200 data 50,20,60,40
2210 data 70,20,80,40
```

2220 rem ZONE CO-ORDINATES FOR ROOM FIVE

2230 data 10,20,20,40

2240 data 30,20,40,40

2250 data 50,20,60,40

2260 data 70,20,80,40

In this routine, we have five rooms in the maze, each with four exits which all lead to other parts of the maze. The MAP array works like this: the variable ROOM holds the number of the player's position in the maze, in other words, the room number. The first set of data statements allow us to specify which exit leads to which room. For example….

10 read MAP(1,1),MAP(1,2),MAP(1,3),MAP(1,4)

20 data 2,3,4,5

This means that exit one leads to room two, exit two leads to room three and so on. When you run this routine you will see four boxes on the screen, representing four zones, or exits in this case. Clicking the left mouse key

in a zone will set the variable ROOM to the new destination room. With this method, you can easily tell where you are just by reading the ROOM variable. With this, you can call up the part of the maze you want.

100 screen$(logic,16,0)=R$(ROOM)

The arrays XZ1, YZ1, XZ2, and YZ2 hold the coordinates of the exit boxes on the screen. The format goes:

XZ1(ROOM,EXIT)

So, we can set up four zones in the present room using the SET ZONE command then check which zone the player chooses. The exit zones could be four arrows pointing in four directions. Using this method, you can check what’s on-screen in this method. For example, if the player clicked on a baddie to fight him, then you could say that the baddie is in room three and is in zone two, and check like this.

10 repeat

20 CH=zone(0) : wait vbl

30 until CH<>0 and mouse key=1

40 if ROOM=3 and CH=2 then gosub 1000

Where line 1000 onwards holds the routine for fighting the baddie.

BANK STACKING

Think of a large box, this allows us to put more than one item into it. It’s the same with banks, we can set the size of it then load the files into it one after another. This method with stack PAC pictures.

First make a note of the length of each PAC file then add them all together, then add about twenty bytes and make it an even number. You then choose a bank and reserve it to this size.

10 reserve as data 5,80000

Note reserving a work bank makes it a temporary bank while reserving a data bank means it can be saved along with your program.

Load the first file in like this.

20 load”pic1.pac”,start(5)

Now the next thing to do is load the next picture into the position where the last picture ends. So, get the length of the first file and make it into an even number. For example, if the file is 1787 bytes long then call it 1790 bytes long. We can now load the second file in, in front of the first file just like this.

30 load”pic2.pac”,start(5)+1790

After this, we just take the length of the pictures already loaded and load the next picture in. For example, let’s say our pictures are like this.

PIC1.PAC LENGTH 1787 ROUNDED TO EVEN 1790

PIC2.PAC LENGTH 2136 ROUNDED TO EVEN 2140

We take these two values and add them together, which makes 3930. If the value is an odd number then you must make it even. We can then load this file in.

30 load “pic3.pac”,start(5)+3930

Do the same with the other files already loaded, add them together and round them up to even numbers to find where to load the next picture in. To get at them, all you have to do is this.

40 unpack 5 : rem unpack the first picture

50 unpack start(5)+3930 : rem unpack picture three

There is an easier method if you have the missing link extension. This has two commands called BANK LOAD and BANK COPY. Use the MAKEBANK program on the source disk to load all files and save them as an FBANK with the extension BNK. You can then load and use them like this. PICNO is Picture Number.

10 mode 0 : key off : flash off : curs off

20 PICNO=1

30 reserve as work 5,80000

40 bload”pics.bnk”,5

50 reserve as work 6,9000

60 bank copy start(5),start(6),PICNO

70 unpack 6

This copies the specified picture from the stacked bank into bank six and unpacks it to the screen. You can use this method with the smaller files but it tends to corrupt some larger ones. The first method however should work with all files.

In case you were wondering, the BANK LOAD command allows you to load a file from a stacked bank on a disk.

STOS CODE TO AMOS CODE

As STOS and Amos are similar languages it was easier to port the STOS code of some of my games to Amos rather than rewrite it. In this article, I will show you how to port your code over to Amos.

Converting STOS source code to Amos is easy. Simply save it as an ASCII file with the command save"game.asc". Then port it over to your Amiga by formatting a MSDOS disk with Fastcopy 3, making sure you turn the fast format option off. Then it's a simple case of reading the disk in your Amiga's drive using the Crossdos driver which you'll find in the Devs drawer on your workbench disk. Simply double click on the icon to activate it. Once it's on disk then you can simply load the ascii file into Amos with the Load Ascii option.

Now a thing to remember is although Amos has the most normal STOS commands, there are still some differences. Let's take this line.

```
10 key off : mode 0
```

In Amos, there are no such commands. You should change this to…

```
10 Screen Open 1,320,200,16,Lowres
```

This does the same job as the command 'mode 0'.

When you list your source code you will notice that Amos will show commands it doesn't understand in capital letters. You just need to find the Amos versions of those commands and change them around.

Sometimes there are problems with loading in certain files with long lines and you will get a "Line too long" message. All you have to do is load the ASCII file into a word processor and make the lines shorter. For example:

```
10 for X=1 to 20 : XB(1,X)=FRNE+39*20/200 : next X
```

This can be set out as…

```
10 for X=1 to 20
20 XB(1,X)=FRNE+39*20/200
30 next X
```

If you have a lot of conditions in an If Then statement then you can use an Amos feature which allows you to have as many conditions as you want. The following routine will show you what I mean.

```
10 If CHOICE=1
20 Screen Copy 5 To 3
30 Music 1
40 FE=FE+100
50 Endif
```

Once the source is converted then it’s time to port over and convert your program’s data. Random Access files and Seq files can still be read within Amos but the commands are slightly different. Chipmusic will not play on the Amiga but you can play mods in your game instead. Sample banks I have not found a way to convert but you can simply load single samples into Protracker then save the lot as a mod, then save them out when you port the mod over.

Pictures can be loaded into STpaint and saved as IFF files for the Amiga. Sprite banks can be converted to pictures using the Put Sprite command to fill a picture screen with sprites.

Final note: Amos uses both line numbers and labels so there is no need to change that unless you want to.

STOS COMPILER ERRORS

The STOS compiler is a very useful program to have because it does the following things.

It speeds STOS programs up to three times their normal speed.

It saves the program as a single PRG file so there's no need to copy the STOS folder onto the copy disk.

Compiled STOS programs are smaller than normal ones, saving disk space.

There's no way a PRG file can be changed back into a BAS file so no one can hack into it.

Even though the compiler seems like a great piece of software, it does have its issues. For example, when you compile a program to disk and then load it, the Function Key window flashes twice before the program starts. The way around this problem is to remove these commands from the program before you compile them.

key off

mode 0

hide on

curs off

From the Compiler's main screen, go to the options menu and set the MODE option to LOW. Next, turn off the other commands by clicking on the OFF boxes. Compile the program and the screen appears blank when the program is loaded. Works better when the compiler sets the commands.

If you are pressed for space, then reduce the Sprite Buffer to 500 bytes. If the mouse pointer is not used in your game then select the NONE option in the options menu, but if your program just uses low-res pointers then turn off the LOAD MED/HI option instead.

The Compiler has a couple of problems on the STE, first, trying to set it to change to Medium Rez results in the compiled program to run in the rez it was loaded in. Change the mode option to medium.

The second problem is with the STOS Tracker. On the STE the compiler always tells you that the last line of your program has an error, no matter what it is. To fix this you need to update the Compiler with the program on the Tracker disk called COMP207.PRG updates the Compiler to v2.07 so the tracker commands will compile okay.

The compiler will sometimes list a line in your program and report an error, even though the line is okay. The reason it does this is that it is having trouble compiling it. This can happen on long lines of code and the FOR/NEXT loops. Look at this example.

10 pen 6: for x=1 to 10 : print”Hello” : next x

The compiler doesn't like this line because of the "pen" command, why is a mystery, but remove the "pen" command and everything is fine. When using the FOR command, make sure it's the first command on the line, like so…

10 for x=1 to 10 : pen 6 : print"Hello" : next x

See how I have put the "pen" command inside the loop? This lead to another problem I found. Look at this example.

10 box 10,10 to 100,20

When this line is found in a compiled program, the box is drawn in the current paper colour. The solution is to add an "ink" command at the start of the line.

10 ink 1 : box 10,10 to 100,20

The compiler's main problems lie in the basic listing, it will report any lines it doesn't understand. Normal errors would be spelling mistakes or the wrong use of a command. But sometimes it just has trouble compiling it in this case just play about with the listing and re-word it a bit, then try again.

In the COMMANDS.BAS file, there is a list of words that cannot be used in a compiled program. Most of them are Direct Commands so they can't be put on a line anyway. The "run" command can't be used so if your program requires a BAS file to load and run later in the program then you'll just have to add it to the end of the program and call it using the 'goto' command.

Also, if your program has LOAD"name.var" in it, it will not work when compiled. The compiler will compile it but prints an error in the compiled program when it comes across this command. The solution is to save your variables as SEQ or RANDOM ACCESS files. For example:

10 open #1,"R","varible.dat"

20 put #1,N$

30 close #1

You can then load the contents of N$ like this.

10 open #1,"R",varible.dat"

20 get #1,N$

30 close #1

Despite these issues, most programs should compile fine. One other important thing to remember is that extensions have a compiler version with the extension .ECA. Make sure you have that installed.

STOS EXTENTIONS

In this article, I hope to shed some light on what a few people seem to be having problems with…. STOS extensions. Various questions are raised on the subject, what is an extension, what does it do, how do I install it, and how is one written. I shall attempt to answer these questions.

1> WHAT IS AN EXTENSION?

When Francios Lionet developed STOS, he decided that there could always be room for improvement. In other words, fix it so extra commands could be added. So, instead of releasing a new version of STOS with new commands, he fixed it so that the new commands could be added.

When we enter a command into STOS, it is not understood by the ST in its basic form. It first has to be translated into a language that the ST can understand, which is machine code. Suppose we type:

plot 320,100,1

The assembly routine for this command would look like this.

move.I #1,-(a6)

move.I #100,-(a6)

move.I #320,-(a6)

jsr plot

If we look in the STOS folder, we will see a large selection of files. As STOS loads – each of these files are loaded into memory, each file contains machine code routines for each basic command entered into STOS. For example, if we entered the above 'plot' command, STOS would look through the BASIC.BIN file for the machine code routine for it.

STOS was written in assembly language then assembled into machine code, which is a list of binary numbers. So the SPRITE.BIN file contains the binary numbers for the SPRITE commands. So as we can see, STOS is not just one large piece of code, it's split into different parts. If we remove the FLOAT.BIN file then we wouldn't be able to use floating-point numbers as STOS doesn't know the machine code routine.

So, a STOS extension is just an extra file containing the command names and machine routines like the .BIN files which are first written in assembly then translated to machine code. STOS loads it into memory and when we enter one of the new commands, STOS looks in the extra file and finds the machine code routine for it, then executes it.

Sounds confusing, doesn't it. Well, to use an extension this information is not important anyway so panic ye not.

You've likely been using an extension without realising it, STOS already has one installed in the later versions. The COMPACT extension gives us two extra commands PACK and UNPACK. Load STOS and enter the following routine….

10 key off : mode 10 key off : mode 0

20 reserve as screen 5 : reserve as screen 6

30 load”pic.pi1?,5

40 pack 5,6

50 unpack 6

This routine will load a degas picture into bank five and the “pack” command will compress it to a smaller size then put it into bank six. The ‘unpack’ command will expand the compressed picture from this bank and copy it to the background and physic screens. Now, save this routine to disk and exit STOS by typing “system” to go to the desktop.

Insert the STOS disk and open the STOS folder, look through the files for one called COMPACT.EXA. This is the file that holds the new command names along with the machine code routine for each one. This is the compact extension for STOS, the interpreter version.

If we look at the three letters following the dot, we see it says EXA, this informs STOS that it is extension A. As STOS loads it reserves a slot in memory for this file and names it slots A, so when it comes across a command from this file, it looks in slot A for the command’s information and runs it.

Let’s try something: rename EXA to XXX and re-load STOS. Next, load the compress routine we did earlier and list it. It now looks like this…

10 key off : mode 10 key off : mode 0

20 reserve as screen 5 : reserve as screen 6

30 load”pic.pi1?,5

40 extension #A 5,6

50 extension #A 6

What’s happened? Where are the PACK and UNPACK commands? What’s happened is since we renamed the EXA part of the file STOS hasn’t loaded the file into memory (slot A), the routine has told STOS that the command names and routines are in the COMPACT.EXA file but as it’s not been loaded, slot A is empty. So STOS lists the PACK and UNPACK commands as it has in lines 40 and 50 telling us that extension slot A is empty. We can see this by running the program.

Type ‘run’ and the following will appear……

Extension not present in line 40

40 extension #A 5,6

So, to get the commands back we need to rename the extension filename back to COMPACT.EXA. Reboot STOS, run the routine again, and hey presto, the commands

appear back in the listing. This is because the extension details are sat back in slot A waiting to be used.

2> HOW DO WE INSTALL AN EXTENSION?

Before we can use the new commands in an extension, we first need to install the extension. This is very simple indeed; all we need to do is put the extension file in the STOS folder on the STOS disk. When STOS loads up, it looks inside the STOS folder and loads each file in it, so inside the STOS folder is the COMPACT.EXA file which STOS will load into memory (SLOT A) when it comes across it. The extension file that goes in the STOS folder is called the "interpreter" version.

Some extensions, such as "Misty", have a program supplied to install the extension for us. We just load the program, select the right install option and insert the right disk, other extensions just have the files on disk leaving us to install them ourselves.

Extensions such as "The Missing Link' are cut into two or three different extension files so that means each interpreter file has to be put in the STOS folder for us to use all the commands. Let's look at the COMPACT extension and see what the extension name means.

COMPACT.EXA

E= The file is a STOS extension

X= And it's the interpreter version

A= It is loaded in slot A

If you were installing the 'missing link' extension then you would put these files in the STOS folder……

LINK1.EXQ

LINK2.EXR

LINK3.EXS

As you can see from this last example, each of these extension files has a different SLOT letter…Q, R, S. If you have two extensions installed that have the same SLOT letter then only one will load. STOS loads each letter in alphabetical order, so it will load extension A, ignore the next extension with the same SLOT letter and proceed onto the next extension file it finds. As far as I know, three extensions use the SLOT letter S, these are STOS Tracker, Link3, and STOS 3D. We can't change the SLOT letter as the file will only load into its original SLOT number, but we can stop it from loading by renaming the file extension, IE: the three-letter name of one file to XXX.

THE STOS COMPILER

If you have a copy of the compiler then you need to know that before you can compile a routine using new commands then you need to install the compiler extension into the COMPILER folder on the COMPILER disk.

When we compile a program, the compiler looks for each basic command it finds in the routine and converts it to machine code by looking at the files in the COMPILER folder and putting the commands information into the compiled program. So for our new commands to be compiled we need to put the COMPILER version of the extension in the compiler folder. Have a look in this folder and you will see this file.

COMPACT.ECA

This file tells the compiler that…….

E= The file is a STOS extension

C= And it is the COMPILER version

A= It is compiled into SLOT A

Try loading that routine we prepared earlier and compiling it. You'll see that it works okay, but try changing COMPACT.ECA to COMPACT.XXX and try compiling the routine again. The compiler reports this message…

Extension not found in line 40

This quite simply means that the compiler can't compile the extension because it can't find it on the disk, just rename COMPACT.XXX back to COMPACT.ECA and everything will work fine.

Compiler users will have the compiler's own extension installed and will see it as COMPILER.EXC and COMPILER.ECC.

3> HOW DO I WRITE AN EXTENSION?

Extensions are written in assembly language. A full tutorial is available from the Game Makers Manual by Stephen Hill which you can Download Here along with other STOS manuals.

4> WHERE CAN I DOWNLOAD EXTENSIONS?

A complete list can be found on Exxo's STOS Pages. Each extension loads into the slot specified by the extension but none ever use slot B as it has a bug…

USING THE STOS SPRITE, MOVE AND ANIM COMMANDS

Sprites are the little graphical images that move about on-screen such as ships, aliens, bullets, little men, etc. Let's look at how we can do this.

THE SPRITE COMMAND

Before you can use the sprite command you will need some sprites in memory. For example, try loading the animals1.mbk sprite bank from the STOS disk.

EG: load"animals1.mbk"

As it's a sprite bank, STOS automatically loads them into bank one. Now let's put it on screen.

```
10 sprite 1,100,100,14
```

This put a monkey sprite on the screen. Now, let's take a closer look at the sprite command.

sprite NUMBER,X,Y,IMAGE

NUMBER is the number of the sprite which ranges from 1 to 15. You can only place 15 sprites on the screen at any one time.

X AND Y are the X and Y coordinates where you want to put the sprite. So in the above example, the sprite is placed 100 pixels across the screen and 100 pixels down the screen.

IMAGE is the number of the sprite from the sprite bank you want to place on the screen. In the above example its image 14 which is the monkey.

Let's try two sprites on the screen.

```
10 sprite 1,100,100,14

20 sprite 2,160,100,14
```

So we now have two copies of the monkey on screen. Try changing the image number and the second sprite will be another one from the bank. Sprites can be designed using the sprite editor accessory. Refer to the STOS manual on how to create sprites.

MOVING A SPRITE:

The sprite can be moved across the screen by using the MOVE command. It goes like this.

```
move X 1,(1,2,300) or move Y 1,(1,2,300)
```

Let's look at the command in more detail.

move DIRECTION,NUMBER,(SPEED,STEPS,PIXELS)

DIRECTION can be X or Y, it tells STOS to move the sprite either X (across the screen) or Y (up or down the screen).

NUMBER is the number of the sprite to move.

SPEED is the speed to move the sprite which ranges from 1 (fast) to 4 (slow).

STEPS tells STOS how many pixels to move the sprite at a time.

PIXELS tells STOS how many pixels to move the sprite.

Look at this example

10 sprite 1,10,10,14

20 move x 1,(1,2,100)

30 move on

So the sprite is moving across the screen at a speed of 1, in steps of 2 pixels at a time, and 100 pixels across the screen. The move-on command tells STOS to start the movement. Note that sprites work on interrupt so your program can be doing over things while the sprite is working.

We can add extra letters to the move command to make it do other things like L and E. Normally the sprite will move and stop at the PIXELS parameter you choose but you can fix STOS to keep moving the sprite along its set strings by adding the letter L at the end of the move command. Using the E command allows you to stop the movement at a certain point.

EG:

10 sprite 1,10,10,14

20 move x 1,"(1,2,100)(1,-2,100)L" :rem keep moving sprite

30 sprite 2,10,50,14

40 move Y 1,"(1,2,100)E50?:rem stop sprite at Y co-ordinate 50

50 move on

ANIMATING THE SPRITE

We can step through the images of a sprite by using the ANIM command. The format goes like this…

anim NUMBER,(IMAGE1,DELAY),(IMAGE2,DELAY) etc

NUMBER is the number of the sprite to animate.

IMAGE is the image number to show.

DELAY is the time to pause between each image.

Let's animate the monkey.

10 sprite 1,100,100,14

20 anim 1,"(14,5)(15,5)(16,5)(17,5)L" : anim on

So this command flicks the sprite between images 14 to 17 and then loops back to the start. We now have a walking monkey.

OTHER COMMANDS

MOVON(NUMBER)

This command checks to see if the sprite is still moving. It returns 0 if it isn't and 1 if it is.

UPDATE, UPDATE ON, UPDATE OFF

STOS updates sprites every 50th of a second. This can be awkward when you want to do other things and STOS is using processor time for its updating. STOS is always checking a sprite and updating its position and you can turn the feature off by using the Update Off command, and Update On can return things to normal. You can then update the sprite when you want freeing some processor time like so…

```
10 update off
20 for X=10 to 100
30 sprite 1,X,10,1 : update
40 next X
```

Try removing the update command from this example and the sprite will not be displayed on the screen. It's still there but just not being copied to the screen.

VARIABLES AND DIMENSIONS

Variables and Dimensions are a way of storing information in STOS to use in your program. The two main pieces of information are words and numbers. Your program would decide if one of these equalled something and act upon it. Here's a simple example of a variable being used.

VARIABLES

```
10 print "Please type in a number"

20 input A

30 print "You choose the number";A
```

What happens here is that the computer labels a little box as 'A' and inserts the number you typed in at the input prompt into it. The number you typed is now stored in the computer's memory in a little box called 'A'. Line 30 looks in the box and prints its contents to the screen, which of course is the number you entered. 'A' is the name we have chosen for the variable and the input command is one way of putting information into it. Another way to put information into a variable is like this.

```
10 let NUMBER=100

20 print NUMBER
```

Line 10 tells the computer to get one of its boxes and stick a label on it and call it NUMBER, then puts the value '100' inside it. Note, the use of the command 'let' is optional and can be removed so your routine would be.

```
10 NUMBER=100

20 print NUMBER
```

Variables can be called anything you wish, using letters, words, and symbols, but you must make sure that there are no STOS commands in the name, or else you get something like this……

```
10 sin GLE=10
```

If you used the word 'SINGLE' then STOS finds the command 'sin' and gets confused. Try different names until you find one that appears in capitals.

Not only numbers can be used in variables, so can words or letters. If you wanted to put a word into a variable then you must add a '$' to the variable's name and the word must be put in between two quotes. For example.

```
10 NAME$="DEANO"

20 print "MY NAME IS ";NAME$


10 input "What is your name";NAME$

20 print "Hello There ";NAME$
```

You can also add variable values together which can be used in a game to add points to your score, for example.

```
10 SC=100

20 print "Your score is ";SC

30 print "Press a Key"

40 SC=SC+10 : goto 20
```

Line 40 tells the variable SC to equal the value of itself which is '100' and the add '10' to itself. The same applies to letters, usually known as strings of characters. For example.

```
10 input "What is your first name";NAME$

20 input "What is your second name";SURNAME$

30 A$=NAME$+" "+SURNAMES

40 print "Hello ";A$
```

DIMENSIONS

As you know, a variable is a little box inside the computer's memory. A dimension is a large box full of little boxes. Think of a cassette box, it's one big box with so many slots for your cassettes. A dimension allows you to store so many pieces of information inside one variable. Here's an example of how one could be used.

```
10 dim A(2) : rem Set up one variable box with two slots

20 A(1)=100 : A(2)=200

30 print A(1),A(2)
```

The same method can be used with strings.

```
10 dim NAME$(2)

20 NAME$(1)="ST" : NAME$(2)="PLUS"

30 print NAME$(1),NAME$(2)
```

Dimensions can be used the same way as variables, the only difference is that with dimensions you use the variable name plus the number of slots in the variable box….IE: dim SC(10).

Another type of dimension is the two-way dimension. This is setting up so many boxes with so many slots. Try this example.

```
10 dim NUMBER(5,10) : rem Set up five boxes each with ten slots.

20 NUMBER(1,1)=100 : rem Put '100' in box one, slot one

30 NUMBER(2,1)=500 : rem Put '500' in box two, slot one.
```

```
40 print NUMBER(1,1)

50 print NUMBER(2,1)
```

This is useful for directions in adventure games. For example, this routine sets up a dimension with five locations and four exits.

```
10 dim D$(5,4)

20 for X=1 to 5 : For Y=1 to 4

30 read D$(X,Y)

40 next Y : next X

50 data 2,3,4,5

60 data 1,2,3,4

70 data 5,2,4,3

80 data 7,6,2,1

90 data 2,4,1,2
```

I recommend you use clear variable names and plenty of REM statements in your program so you can keep track of what each variable and dimension does.

WRITING A SAC ADVENTURE GAME

For you to use this tutorial you will need a copy of the STOS adventure Creator (SAC) which you can download on my website (deanosharples.com).

In this tutorial, I am going to show you how to write a full adventure game called Witches Castle using SAC. The story goes like this: The wicked witch has kidnapped the princess and is holding her captive in the castle. A warrior has been charged with the task of defeating the witch, rescuing the princess then escaping the castle. So we have the plot and the main character, so we shall define an adventure game world which is the castle. Let's make a note of the locations we would find in a castle, the list would look like this:

Castle entrance

Main hall

Dungeon

Throne room

Musty room

Of course, a castle would have more locations than this but I'm just keeping it short and simple. Now let's add two outside locations.

Lake

Enchanted Forest

In this game, the player would start at the castle entrance and these two locations can be on either side of him while the castle is in front of him.

Now we need some objects which are as followed:

A Steel Key….. used to unlock the dungeon door

An Axe……. To kill the rat

Guards Uniform… Has to be worn to get past the guards

A Goblet… Contains holy water which kills the witch

Now we've worked this out, we can enter the information into Sac. Load the creator, go to the location menu and insert the following seven locations.

Location 1

You stand before the mighty castle which is north. On the top of it is an ugly stone gargoyle whose stare sends a shiver up your spine. You can also go west and east.

Location 2

You are beside a beautiful lake that glimmers under the morning sun. It is surrounded by clusters of trees and bushes which glow brightly. You can only go east.

Location 3

You find yourself in the Enchanted Forest which is dark and creepy. The trees and bushes have no beauty and make further movement impossible except west.

Location 4

The main hall of the castle. It has a certain mist of evil about it and full of old junk. To the east stands an arched doorway and west a dungeon. You can also go north and south.

Location 5

You find yourself outside the castle dungeon and all you see is a large steel door. The hall is east.

Location 6

Emerging into the Throne Room you see nothing but murky cobwebs and a small steel throne. You can see a large cauldron and an arched doorway leading west.

Location 7

You are in a bare musty room. A doorway is south.

Now go to the connection definer and insert these connections.

Location 1

West to location 2, East to location 3

Location 2

East to location 1

Location 3

West to location 1

Location 4

South to location 1, West to location 5, East to location 6, North to location 7

Location 5

East to location 4

Location 6

West to location 4

Location 7

South to location 4

Now enter the objects in the object definer along with their locations and object words.

Object 1…a steel key., location (not created), key

Object 2…an axe., location 3, axe

Object 3…a guards uniform., location 2, uniform

Object 4…a goblet., location (not created),goblet

Go straight to the Message definer (m) and enter these messages.

Message 1

The water melts the witch into nothing and drops the key.

Message 2

As you are wearing the uniform, the guards lower the drawbridge to let you in.

Message 3

You kill the rat and a goblet of holy water appears.

Message 4

The witch is here, she is holding a steel key.

Message 5

You unlock the door and the princess runs out and gives you a big hug then says C'mon let's leave this castle.

Message 6

There is a vicious rat here.

Message 7

The princess is here.

Message 8

Well done, you have rescued the princess and completed your task.

Message 9

The guards refuse to let you in the castle.

If you wish, you can add some examine messages to the game, this involves four messages in the Examine Object option and as many as you want in the Examine Location option.

Now we have all the data we need for now. if you wish to add more later you can always load it back in.

Now save your data file, load Stos, and load the Editor program.

As you may have guessed we have a rat and a witch in the game as well as a princess so let's define them as variables.

WITCH=1 : PRINCESS=0 : RAT=1

So at this moment the witch and rat are alive and the princess is not yet rescued. Add these lines as H_P events

if LOC=1 and OB_LOC(3)<>W0RN then print MESSAGE$(9)

if LOC=1 and OB_LOC(3)=W0RN then print MESSAGE$(2) : MAP(1,1)=4

if LOC=6 and WITCH=1 then print MESSAGE$(4)

if LOC=7 and RAT=1 then print MESSAGE$(6)

if MEET=1 then PRINCESS=LOC : print MESSAGE$(7)

if LOC=1 and PRINCESS=LOC then print MESSAGE$(8) : goto 2610

The first event tells Stos that if the player is not wearing the uniform the guards won't let him in and the second event tells Stos that the guards will let him in because he's wearing the uniform. The variable W0RN holds the location of worn objects which is 1000. Now add these L_P events.

If WRD$(1)="throw" and WRD$(2)="goblet" and OB_LOC(4)=CARRIED then OB_LOC(4)=NC : OB_LOC(1)=LOC : print MESSAGE$(1) : NO=1 : WRD$(1)="" : WRD$(2)="" : WITCH=0 : DONE=1

So when the goblet is thrown, it vanishes to the Not Created location, and the key is created at put at the player's location.

If WRD$(1)="kill" and WRD$(2)="rat" and OB_LOC(2)=CARRIED then RAT=0 : OB_LOC(4)=LOC : print MESSAGE$(3) : NO=1 : WRD$(1)="" : WRD$(2)="" : DONE=1

This line kills the rat and creates the goblet.

If WRD$(1)="unlock" and WRD$(2)="door" and OB_LOC(1)=CARRIED then print MESSAGE$(5) : MEET=1 : NO=1 : WRD$(1)="" : WRD$(2)="" : DONE=1

This line rescues the princess.

Well, that's it, the game is finished, all you have to do is change the loading name from 'game.adv' to the one you choose when you saved the data from the creator. Give it a test run and correct any mistakes you find. When you're sure it works okay then why not add a few extras like samples, music, and graphics.

As you can see, you have just written a full adventure game in just one day, with more practice you could write large more complex games with Sac and give them away if you wish.

Before I go, I will give you one last example of how the player can fight monsters that can take more than one hit. Let's say the player can take 10 hits and the monster can take 5, all you have to do is set and name two variables.

MONSTER=5 : PLAYER=10

Then using the method described in chapter 5, use a L_P event to take a point from each variable every time then use an H_P in the L_P section to check if any of the variables equal 0 and if so, use a message to tell the player that either he or the monster is dead and either end the game or give him some points.

That’s the end of the tutorial. If you write any adventures using Sac then please let me know. I would love to see them.

FINAL WORD

Out of all the computers and languages I've used – STOS and the STE remain the most special time of my life. The many hours I spent trying to get some code to work how I wanted it to and all the hours spent writing tutorials have all been worthwhile because I can look back and enjoy the happy memories of the time.

Happy programming,

Deano Sharples