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 plusing 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.