Laser Basic

Amstrad CPC
464/664/6128

ocean

CONTENTS

LASER BASIC EXTENSION

by OASIS SOFTWARE

COPYRIGHT NOTICE

THIS MANUAL

Piracy has reached epidemic proportions and it is with regret that we are forced to reproduce this manual in a form which cannot be photocopied. Our apologies for any inconvenience this may cause to our genuine customers. A reward will be paid for information leading to the successful prosecution of parties infringing this Copyright Notice.

NOTE

This manual is essential for the use of Laser Extended BASIC. For this reason we would warn customers to look after it very carefully, as separate manuals will not be issued under any circumstances whatsoever.

ENQUIRIES

If you have any queries on the use of Laser Extended BASIC, please send them to us in a letter, ensuring you enclose the Enquiry Card printed on the last page of this manual. A new card will be returned to you with your reply. Please note that enquiries not accompanied by the card will not be answered.

LASER BASIC EXTENSION
by John Gross

INTRODUCTION

Laser Extended BASIC is an extension to the existing set of BASIC commands in the Amstrad's BASIC ROM. Although Locomotive BASIC is one of the most elegant implementations of the time honoured language, it's features had to be designed to make it as flexible as possible. BASIC has numerous applications but the specific area of our interest is graphics and animation. Laser BASIC was designed to enhance the ease and in particular the speed, with which complex animated graphics could be produced and about 200 commands and functions are included to do this. The extra commands are all implemented as 'BAR' commands (RSX's) and so the first character of each is a vertical bar (shifted @). Where commands are referred to in the text the 'BAR' is ommitted

Those users already familiar with the Lightning series will recognise most of the command set, although for clarity a number of the command names have been changed. Laser BASIC does not produce stand-alone programs (you need the extended interpreter to be resident) but a compiler has also been developed which will make your Laser BASIC programs run faster and not require the extended interpreter to be resident. This will mean that you can market your programs commercially.

The chief aim of Laser BASIC is to make the use of graphics, animation and sound as simple and 'unfussy' as possible. The command set has been designed so that someone with a modest knowledge of BASIC and the minimum of patience can see results as quickly as possible (hence the extensive use of examples). However, it has to be said that if Laser BASIC was merely a couple of dozen very simple animation commands then the more adventurous user might well question his value for money. In fact there are more that 200 commands with some fairly advanced features (tracking sprites and sound handling for instance) and so it is our hope that months from now you will still find the package challenging but not frustrating.

OTHER AMSTRAD LASER PRODUCTS

Although Laser BASIC is a fully self contained package there are a number of companion products which compliment its use. For details of availability and prices contact your local dealer or write directly to us.

The Laser BASIC Compiler

Laser BASIC programs require the extended interpreter to be resident and this limits the potential user base for your completed programs. The Laser BASIC Compiler, however, will compile your BASIC programs into fast Z80 machine code that will run on any Z80 Amstrad micro without the need for Laser BASIC to be resident. Most of the Amstrad's BASIC is catered for but integer arithmetic is used and so none of the floating point functions are supported. It is a good idea to start all your programs with DEFINT A-Z as a matter of course. Not only will this make sure that you have used integer arithmetic, but it will also make your program run faster and use less memory.

The Laser Graphics Designer

Although Laser BASIC is supplied with the Sprite Generator Program, a much more powerful ICON driven screen/sprite designer is available that produces graphics in a format compatible with all the products in the Laser range. Remember that no other designer will produce compatible graphics.

The Laser Music/Sound Composer

Again, Laser BASIC is supplied with a sound generator program which enables music and sound effects to be assembled into sprites. A much more comprehensive package that will ease and speed up the development of music and sound is also available. It is the only package which will produce compatible sound sprites.

The Laser Assembler/Monitor

There really isn't space to do this product justice. We believe it to be the most powerful assembler/monitor ever produced for a Z80 machine. The assembler has a number of compiler like qualities and will generate code for structured programming loops as well as arithmetic and logical expressions. The monitor has some features which have previously only been obtainable by expensive hardware analysers and emulators.

Although the assembler tokenises its text for speed and efficiency, source files from a number of alternative assemblers can be loaded and converted into the Laser assemblers tokenised format.

The Graphics/Sound Source Code

An enormous amount of time and effort went into developing and debugging the graphics and sound routines which Laser BASIC and the other Laser packages use. The advanced user who works in assembly language may be able to make great savings in effort by using these routines in their games writing. The routines are supplied as souce code, in a format compatible with the Laser Assembler and each entry point is carefully documented (the workings of the code itself is not documented) Any source code which is not used can of course be deleted before assembly time and so the final run-time object code is likely to be a lot smaller than that employed by the compiler.

Mini Laser BASIC

A cut down version of Laser BASIC which supports only the main features of Laser BASIC and leaves much more memory for sprites and BASIC source is also available.

GLOSSARY OF TERMS USED IN THIS MANUAL

SPRITES

A sprite is a software controllable graphics character which is stored in memory as a series of data 'bytes'. The block of memory which makes up the sprite can be displayed by 'PUTting' it into video memory where it appears as a visible image on the screen. References to 'sprites' generally mean the bytes of data in memory which make up the graphics character but occasionally apply to the visible screen image. Laser BASIC allows 255 sprites to be defined, each with their own user selectable dimensions (up to 255 by 255). The limit on the size and number of sprites available to the user is set by the amount of memory available.

SCREEN WINDOWS

A screen window is a section of the screen defined by the four variables (see Laser BASIC variables) COL, ROW, HGT and LEN. COL is in the range 0 to 79, ROW is in the range 0 to 199, HGT is in the range 1 to 200 and LEN is in the range 1 to 80. The units for COL and LEN are bytes - 1/2 characters in 4 colour mode and 1/4 characters in 16 colour mode. The units for ROW and HGT are pixels in both modes. COL and ROW specify the column and row position on the screen of the left hand corner of the window, with ROW 0 at the top of the screen and COL 0 on the far left hand side. HGT and LEN define the dimensions of the window.

To see an example of a window on the screen type in the following line and hit ENTER.

```
|COL,8:|ROW,8:|LEN,32:|HGT,64:|INVV
```

SPRITE WINDOWS

A sprite window is a section of a sprite defined by the variables SPN, COL, ROW, HGT and LEN. SPN specifies the sprite, COL and ROW specify the column and row within the sprite and HGT and LEN define the size of the window.

If the window defined by these variables lies outside the sprite or overlaps its borders then the command will not execute but no error message will be issued.

SPRITE SPACE

Sprite space is the area of memory containing all previously defined sprites. The top of sprite space is 28671 decimal (6FFF HEX) and the lower end grows downward from this point.

SCREEN OPERATIONS

These are operations which are carried out on a particular area of the screen. The area of the screen to be operated upon is called the screen window and has been defined in an earlier section. The operations themselves include scrolls, inversions, reflections, enlargements etc., and most commands in this category have a suffix 'V' for 'video', eg. MIRV, MGXV and so on. If the window overlaps the edge of the screen then it will be 'clipped' to lie 'on-screen'.

SPRITE OPERATIONS

These cover more or less the same operations as the screen window commands but this time a sprite is operated upon in memory instead of a section of the screen. The only variable used is SPN and the syntax for these commands is the same as those governing screen operations with an 'S' replacing the 'V'. The result of these operations can only be seen when the sprite is re-displayed to the screen using a PUT or MOVE. command.

SPRITE WINDOW OPERATIONS

These operations are carried out on a section of a sprite in memory and more or less the same facilities are available as for the former two sets of operations. This time the commands have the suffix 'P' and the section of the sprite to be operated upon is defined by SPN, COL, ROW, HGT and LEN. The sprite window is 'clipped' to lie 'on-sprite'.

SPRITE/SCREEN OPERATIONS

These are operations between the screen and a sprite. The dimensions of the sprite are used as the dimensions of the screen window and COL and ROW are used to give the co-ordinates of the top left hand corner of the window, thus the operations are defined using the variables SPN, COL and ROW. If the window lies partially 'off-screen' then it will be 'clipped' so that part of the sprite will be 'PUT' or 'GOT'. Commands in this category are prefixed with 'PT' or 'GT' eg. GTBL, PTXR, PTND etc.

SPRITE/SPRITE WINDOW OPERATIONS

These are operations between a whole sprite and a window within a second sprite. The two sprite numbers are held in SP1 (the sprite not containing the window) and SP2 (the sprite containing the window). The dimensions of the window are the dimensions of the sprite not containing the window and the position of the window in the sprite whose number is held in SP2 and is specified by SCL and SRW. The window will be 'clipped' if it overlaps sprite SP2. Commands in this group are prefixed with 'PM' or 'GM'.

SPRITE WINDOW/SCREEN OPERATIONS

These are operations between a screen window and a sprite window. As before, ROW, COL, HGT and LEN define the screen window, but this time SCL and SRW are used to define the position of the window within the sprite. Again the window will be clipped if it overlaps the sprite or the screen. Commands in this group are prefixed with 'GW' or 'PW'.

DUMMY SPRITE

A dummy sprite is a sprite which does not contain data for display. It may be used, for instance, to store a machine code subroutine, an array, a sound program, or may be used as part of a collision detection routine.

INK NUMBERS

The Amstrad has a palette of 27 colours but in 4 colour mode only 4 of these are selected for display and in 16 colour mode only 16 are selected. The colour number refers to one of the 27 colours and the INK number refers to one of the 4 or 16 INK numbers. Each of the INKs can be assigned one of the 27 colours or indeed all of the INKs could be assigned to the same colour, but this is seldom of any use. Laser BASIC uses two variables, IK1 and IK2 which are set to hold an INK number NOT a colour number.

PIXELS

The screen on which images are displayed is divided into a grid of 'PIXELs'. All characters and sprites are made up of pixels. In 4 colour mode the grid is 200 high and 320 wide so it consists of 62,000 individual pixels. Each of these contains a pixel which is displayed with the colour held in one of the 4 INKs. Each of these 62,000 cells can be thought of as holding 0,1,2 or 3. If it contains 0 it will display the colour which was assigned to INK 0 and so on. In 16 colour mode the grid is 200 high but only 160 wide so there are only 32,000 individual pixels. These pixels are no longer 'square' but are twice as wide as they are high. Each pixel can now contain a number in the range 0 to 15 and can therefore display one of the 16 colours which were assigned to the 16 INKs. It is important to understand the difference between an INK and a colour or the remaining manual may well be very confusing, so read this definition and the one previous to it again if you are at all uncertain.

SOURCE DATA

Several groups of commands in Laser BASIC are involved in moving data between sprites and the screen. The source data refers to the data that is actually being moved.

TARGET DATA

This refers to the data that was originally held in the area of memory or screen, into which, the source data is being moved.

TAPE/DISK MAP

TAPE LOCATION	TAPE/DISK FILENAME	DESCRIPTION
Tape 1 - Side A	"LB"	The Laser BASIC extension
	"SPT2SPR"	The example sprites used in the example programs
Tape 1 - Side B	"DEMO"	The Laser BASIC extension (modified)
	"DEMO"	The demo program which automatically loads its sprites when run
	"SPT3SPR"	The sprites used by the demo program
	"SPT4SPR"	
	"SPT5SPR"	
Tape 2 - Side A	"SPTGEN"	The sprite generator program
	"SPT1SPR"	The arcade sprites
Tape 2 - Side B	"SNDGEN"	The sound generator program
	"MUSICSPR"	The example tunes

OPERATING INSTRUCTIONS

The Laser BASIC Extension

(i) To load type RUN"LB followed by ENTER.

(ii) Laser BASIC will load, display the copyright message and wait for a key to be pressed.

The Demo Program

(i) Type RUN"DEMO - for the tape version

or Type RUN"LB - press any key, and then RUN"DEMO for disk.

The Sprite Generator Program

(i) Laser BASIC must first be loaded using the previous procedure.

(ii) The sprite generator can then be loaded and executed using RUN"SPTGEN followed by ENTER. Laser BASIC will need to be loaded from Tape 1. All sprites currently in memory are lost and the default maximum sprite number is 120.

NOTE: Ensure the keyboard is set to upper case before RUNning (if not press CAPS SHIFT)

The Sound Generator

(i) Laser BASIC must first be loaded using the previous procedure.

(ii) The sound generator can then be loaded and executed using RUN"SNDGEN followed by ENTER. Laser BASIC will need to be loaded from Tape 1. The sound generator can be 'broken out of' and re-run from any position.

NOTE: Ensure the keyboard is set to upper case before RUNning (if not press CAPS SHIFT)

GETTING STARTED WITH LASER BASIC

First reset the machine by pressing SHIFT, CTRL and ESC simultaneously. To LOAD the Laser BASIC extension just type RUN"LB and press ENTER. If you are loading from tape you will need to press PLAY on the cassette recorder. The program will load and display the copyright notice. Now press any key and the screen will clear and "Ready" will appear in the top left hand corner. For this tutorial session we're going to use the sample sprites, so if you are using tape do not rewind. Now type

```
A$="SPT2SPR":|GSPR,@A$
```

The sample sprites are now loaded and we're ready to start. To be safe, stick to the examples in the text for now and don't type any other Laser or Locomotive BASIC commands.

LASER BASIC VARIABLES

Before going any further it is worth briefly mentioning the way that Laser BASIC deals with its own variables. These are distinct from the normal variables and a full list is given in the more detailed section 'Laser BASIC's dedicated variables'. Since we're going to begin by looking at SCREEN OPERATIONS we'll introduce just 5 variables - COL, ROW, LEN, HGT and SET. Laser BASIC has 16 'SETs' of variables and each SET consists of 17 variables. Let's begin by defining a screen window using SET 0.

```
|SET,0:|COL,24:|ROW,64:|HGT,64:|LEN,8
```

We have selected SET 0 and will continue to work with SET 0 until we change to another SET. Let's begin by taking a look at the screen window we've selected. To do this type:

```
|INVV
```

To begin with the screen window was empty but what we've done is to 'invert' it. This means that

pixels set to INK 0 are now set to INK 3
pixels set to INK 1 are now set to INK 2
pixels set to INK 2 are now set to INK 1
pixels set to INK 3 are now set to INK 0

The window was originally empty (all pixels were set to INK 0) so by executing INVV (invert video) what we've done is to set all pixels to INK 3 which is red. If we repeat the operation by typing:

```
|INVV
```

The window disappears because all the pixels return to their original INK 0 colour. So lets repeat it again so that we can see the window again - type:

```
|SCLS:|INVV
```

The window is now red again. Let's define a second window inside the original window - type:

```
|SET,1:|COL,26:|ROW,80:|HGT,32:|LEN,4
```

We have now defined a second window inside the first and to see it type:

```
|INVV
```

The second window can now be seen clearly because all pixels have been inverted from INK 3 to INK 0 again. Lets now set the pixels in this second window to INK 1. Remember that we are currently working with SET 1. To set the pixels to INK 1 - type:

```
|IK1,1:|STCV
```

Note that the STCV command (set colour video) uses a new variable IK1 to contain the INK number to flood the window with. Let's do something slightly more interesting now - type:

```
|SET,0:FOR I%=1 TO 500:|INVV:NEXT I%
```

This will invert the window defined by SET 0 (which physically contains the window defined by SET 1) 500 times. Before moving on to something more useful let's look at one more example of the use of variable sets. Type the following:

```
|SCLS:FOR I%=1 TO 50:|SET,0:|INVV:FOR J%=1 TO 10:|SET,1:|INVV:NEXT
J%:NEXT I%
```

The inner (J% loop) inverted the smaller window defined by SET 1 ten times. This was contained in the outer loop (I% loop) which began by inverting the SET 0 window (which contains both windows) then executed the inner loop. This use of variable sets not only makes your programs execute much faster but also compacts your program because only one variable is assigned (the SET) instead of all 4 (COL,ROW,LEN and HGT). We'll leave Laser BASIC variables for now but their use should become apparent as we proceed through further examples.

SCREEN OPERATIONS

We've already looked at one screen operation (INVV) but in fact there are quite a few operations we can carry out on the screen windows so let's have a look at the rest. Let's first make two simple observations and mention a few do's and don'ts.

1. You will notice if you enter a line at the bottom of the screen that it does not scroll up in the normal way. This is because Laser BASIC forces the Amstrad's screen scroll to be a 'software scroll' and not a 'hardware scroll'. This is achieved by using a text window which is the full width of the screen but only 24 character rows high instead of the full 25. Thus the 25th character row is never used for text. Do not get confused between the Amstrad's text windows and the Laser BASIC's screen windows. The text window that Laser BASIC works with uses stream 0 so if you set stream 0 to work with the whole screen then be sure to reset it by typing:

   ```
   |SCLS
   ```

   This will clear the whole screen and set stream 0 to be the current stream with the dimensions Laser BASIC uses. If you have allowed the screen to 'hardware scroll' by using stream 0 or any other stream then always execute an SCLS before using any of Laser BASIC's commands. We won't be doing any hardware scrolling in the sample session so we need not concern ourselves for now.

2. Laser BASIC contains commands which operate in MODE 1 (4 colour mode) or MODE 0 (16 colour mode) but there is no restriction on the use of MODE 2 other than that some Laser BASIC commands will not operate properly in this mode (see 'COMMAND SUMMARY'). Your program can change freely between modes but only certain Laser BASIC commands should be used in MODE 2. The commands "MODE 0" and "MODE 1" are replaced by their Laser BASIC equivalents "ONLO and ONHI" respectively. "MODE 0" and "MODE 1" will still function normally as far as Locomotive BASIC is concerned but Laser BASIC needs the execution of "ONLO" and "ONHI" before it knows that the mode has been changed.

Let's move onto the examples and begin putting a pre-defined sprite onto the screen, not strictly speaking a screen operation, but necessary to allow us to fully appreciate the operations we're carrying out. Type:

```
|SET,0:|SCLS:|COL,40:|ROW,8:|SPN,6:|PTBL:|LEN,28:|HGT,32
```

We begin by selecting SET 0 and clearing the screen. We then set the target for the window to have column 40 (half way across the screen) and row 8 (one character from the top). We then select sprite 6 and place it on the screen with PTBL. Finally, we set a screen window around the sprite with the same dimensions as the sprite (32 pixels high and 28 bytes wide). Note that 32 pixels high means 4 characters high and in 4 colour mode 28 bytes wide means 14 characters wide. Now that we've filled a screen window with some meaningful data we can carry out some operations on it. Also note that the operations we carry out on this data will in no way effect sprite 6 which is still stored in memory. Let's begin by looking at some scrolling. We'll scroll the window by a pixel to the right. In 4 colour mode there are 4 pixels per byte (8 per character). We'll scroll with wrap around. Type:

```
|WVR1
```

Remember that the window is 28 bytes (14 characters) wide and since there are 4 pixels per character we could scroll the whole window back to its original position by repeating the operation 112 times and since we have already scrolled it once - hit ENTER 3 times to move the text cursor down and then type:

```
FOR I%=1 TO 111:|WVR1:NEXT I%
```

In fact we could have achieved the same result more elegantly without using the relatively slow FOR-NEXT loop. Certain Laser BASIC commands, of which the scrolls are an example, can be executed in a machine code loop. There are two types of machine code loop, those which synchronize to frame-flyback and those which don't. Frame-flyback to the uninitiated is something which happens 50 times a second when the 'dot' which produces your TV picture finishes scanning the screen and 'flies back' to start the next scan. By synchronizing your operation to frame-flyback, much smoother movement can usually be obtained. Of course it won't be as fast but 50 times a second is fast enough for most applications. To execute the command in a machine code loop you need to follow the command by two parameters. These can be any legal BASIC expressions. The first parameter defines the number of times the command will execute, whilst the second dictates whether or not the loop will wait for frame-flyback between executions. If the value of the second expression is 0, execution will not be synchronized, but if it has any non-zero value, then it will be. Let's have a look at an example which scrolls our window left, 112 pixels without flyback and then inverts it and repeats with flyback.

```
FOR I%=0 TO 1:|WVL1,112,I%:|INVV:NEXT I%
```

In fact there are 12 screen scrolls in all. The syntax is as follows:

First character:	'W' for wrap or 'S' for no wrap.
Second character:	Always 'V' for video.
Third character:	'R' for right or 'L' for left.
Fourth character:	'1' for 1 pixel '4' for 1 byte '8' for 2 bytes

The commands are:

```
SVR1, SVL1, SVR4, SVL4, SVR8, SVL8
WVR1, WVL1, WVR4, WVL4, WVR8, WVL8
```

Let's have a look at a scroll without wrap. If we scroll our window with no wrap around, we will of course lose the screen window data. Let's enlarge the window toward the right and scroll the sprite without wrap this time in larger steps of 1 byte (half a character in 4 colour mode, 1/4 character in 16 colour mode). Type:

```
|SCLS:|COL,0:|LEN,56:|PTBL:|SVR4,56,1
```

Note that repeating SVR4 56 times was enough to scroll the sprite right out of the window because each scroll was by 1 byte (4 pixels). The movement was much faster this time, and because the sprite was much larger it seemed to flicker half way down. This is because the 'dot' arrived halfway through the scroll and this will be explained more fully later on. Before we can proceed any further we'll need to PUT the sprite onto the screen again. This time let's PUT 2 sprites into our window since it is now 56 bytes wide. What we'll do is PUT the sprite into the left hand side of the window, scroll the sprite right and PUT the sprite again so we will twice fill the window with two copies of the sprite. Type ENTER 5 times to move the cursor and then type:

```
FOR I%=1 TO 2:|PTBL:|WVR8,14,1:NEXT I%
```

Note that WVR8 moves data by 2 bytes so it needs to be repeated 14 times to make 28 bytes (4 characters in 4 colour mode) of space for the sprite to be PUT in.

As well as the horizontal scrolling of the previous examples there are also 2 commands for scrolling vertically. WVVN and SVVN scroll vertically with and without wrap around respectively. At this point we introduce a new variable, NPX (number of pixels). If NPX has a positive value then data scrolls upward and if NPX has a negative value it will scroll downward. Again these commands will execute only once if there are no following parameters or will repeatedly execute if followed by 2 parameters. Type:

```
FOR I%=16 TO -16 STEP -1 :|NPX,I%:|WVVN,8,1:NEXT I%
```

This example will begin by scrolling the screen window upward with wrap. The rate of scrolling will slow and eventually begin downward. The downward scrolling again accelerates and then halts, leaving the window contents as they were before the operation began.

Let's move onto some more unusual screen operations but continue to work with the same window. Let's begin by looking at X-expansion. MGXV will magnify the left hand half of the screen window such that it fills the whole window - type:

```
|MGXV
```

Notice that the data in the right hand half has been replaced by the expanded data from the left hand half and has been lost. We can, of course, repeat the operation as many times as we wish and the window will continue to expand whatever is in the left hand half so let's again type:

```
|MGXV
```

The original image of the sprite has been magnified by a factor of 2 and now fills the window. Likewise we can carry out the same operation in the vertical direction - type:

```
|MGYV
```

This time the top half of the window has been vertically expanded to fill the whole window. Again the data in the lower half is lost and again we could repeat the operation as many times as we wanted.

Now let's clear all the data in the window before looking at some other screen operations - type:

```
|CLSV
```

This command clears all the data in the window to INK 0. Before going any further let's PUT the sprite onto the screen again and contract the window to be the width of the sprite again - type:

```
|SCLS:|PTBL:|LEN,28
```

and then press ENTER 5 times.

This clears the whole screen, takes the cursor to the top left, which would over-write the sprite so it is moved down 5 lines.

Let's look now at the mirroring commands - type:

```
|MIRV
```

You will see that the window has been mirrored in the left to right sense. We can, of course, do the same thing in the vertical direction, to do this type:

```
|FIPV
```

The window has now been mirrored in the top to bottom sense. Before moving on let's widen the window and repeat the operation twice - type:

```
|MIRV:|LEN,56:|MIRV
```

Notice that no data is actually lost because all the data in the left of the current window went to the right and vice-versa. There is in fact a second type of mirroring operation which reflects data from the left half into the right half but does not reflect data from the right half into the left half. In effect it produces a symmetric window from half of the image. The best way to demonstrate this is by example, so type:

```
|SCLS:|PTBL:|LEN,14:|MORV:|LEN,28:|MORV
```

and again, press ENTER 5 times.

After clearing the screen, the sprite is placed into the left hand quarter of the window. The window length is then set to 14 and MORV makes a mirrored copy of the left half into the right half of the 14 byte wide window. The length is then increased to 28 and the operation is repeated with the displayed result. In fact we can do the same thing in the vertical direction using FOPV. To see the result - type:

```
|FOPV
```

You will see that the final result is symmetric in the top to bottom sense. Combining these commands with the scroll commands can produce some interesting effects. Type in the following:

```
|SET,1:|COL,14:|ROW,8:|LEN,14:|HGT,16
```

This sets up a window in the top left of the window defined by SET 0. Now type:

```
|NPX,2:FOR I%=1 TO 1024:|SET,1:|WVL1:|WVVN:|SET,0:
|MIRV:|MORV:|FOPV:|INVV:NEXT I%
```

This produces a kaleidoscope effect which could, of course, be greatly improved upon and this is left as an exercise for the user at some later stage. For now let's move onto some more screen operations. Let's look at SPNV which spins the current screen window by 90 degrees in a clockwise sense. We need to introduce two new variables here, SCL and SRW, which are generally used to specify the column and row position of a sprite window within a sprite but also serve in this one instance as the column and row of the target for the rotation. Let's rotate the window we've been working on and send the result to the right of it - type:

```
|SCLS:|PTBL
```

then press ENTER 5 times and type:

```
|SCL,72:|SRW,8:|SPNV
```

SPNV is the one command which only executes in 4 colour mode because the oblong nature of 16 colour mode pixels makes it pointless. It should also be noted that the height of the window being spun is rounded down to the nearest multiple of eight pixels.

This covers all the screen window transformations but there are three other screen operations to mention here. These are concerned with colouring in areas of the window. Let's first look at FILL. This is a slightly unusual FILL utility and uses a slow but memory efficient algorithm to do it's filling. Again the best way to demonstrate it's use is by example - type:

```
|SET,0:|SCLS:|COL,20:|ROW,100:|HGT,50:|LEN,40:|INVV
|COL,22:|ROW,104:|HGT,42:|LEN,36:|INVV
LOCATE 13,16:PRINT"THE FILL COMMAND"
```

It should be pointed out here that FILL takes no notice of screen windows and will fill until either a screen edge is met or a different colour from that at the point at which filling began, is encountered. So if the pixel at the point at which filling began were red then only the red area containing that pixel will be filled. The colour with which the area is to be filled is held in the variable IK1. To FILL the window we have defined we could begin at any point but we'll FILL the background to the sprite and start at the top left of the window. Again we're going to introduce a new variable - XCL, which is the horizontal position, this time measured in pixels and not bytes. For 4 colour mode, column 32 is made up of pixels with X-values of 128, 129, 130 and 131. For 16 colour mode column 32 would be made up of just two pixels with X-values of 64 and 65. This is because whereas each byte represents 4 pixels in 4 colour mode it represents only 2 pixels in 16 colour mode. FILL uses ROW as its vertical co-ordinate and in both modes it is in the range 0 to 199, as it has been throughout. So to FILL the area all we need to do is type:

```
|SET,1:|IK1,2:|XCL,165:|ROW,125:|FILL
```

For those of you that are interested, the rather unusual way that this FILL executes stems from the algorithm used - 'plot anti-clockwise if you can'. It is worth mentioning here that FILL makes extensive use of the machine stack. As a general rule, the more complex the object being filled, the more space required. If insufficient space exists the execution may halt half way through and display **OUT OF MEMORY**.

We now return to a command that is governed by the window defined in the current SET which in this instance should still be SET 0. What we'll do now is to change all the pixels in the window which have INK number 0 to have INK number 2 (this will colour all the areas that FILL didn't get to!). We're introducing another variable, IK2 which is the colour to change to, where IK1 is now used as the colour to change from. The window is already defined so we can set the variables IK1 and IK2 and then execute the command which is SETV - Type:

```
|SET,0:|IK1,0:|IK2,2:|SETV
```

It is important to note that this does not have the same effect as changing the colours held in BASIC's palette because, instead of the colour that an INK will be displayed in, we are actually changing the INK number of particular pixels within the window. Executing this command will often reduce the number of colours displayed in the window and after its execution a window which contained pixels with all four (or sixteen) colours can only contain three (or fifteen) distinct colours. The exact operation of this command will become apparent with use.

Finally the last command in this section is one which we briefly encountered at the start of the chapter - STCV. This will set every pixel in the window to have the INK number held in IK1 which we will set to 2 in this example - type:

```
|IK1,2:|STCV
```

This concludes the introduction to screen operations. You may wish to experiment with the commands you have encountered so far before moving on to the next section which deals with sprite operations.

SPRITE OPERATIONS

Amongst all of the commands we have considered so far there have been none which would effect a sprite in memory. In fact, with two exceptions (FILL and SPNV), all of the operations we have seen can also be carried out on sprites held in memory. This time the result of the operation cannot be seen until the sprite is displayed by 'PUTting' it onto the screen. The sprite operations are similar to the screen operations so we'll give these commands a briefer treatment than their 'SCREEN' equivalents.

The syntax for these commands differs from the previous group in only one respect. Where the screen commands are suffixed with 'V', the sprite commands are suffixed with 'S'. Where the screen window was defined by four variables - COL, ROW, LEN and HGT, the sprite is defined by only one - SPN, the number of the sprite to be operated on. Bear in mind that these commands will alter some of the sprites in memory so after this session you will need to re-load them if you think one of the sprites you wish to use may have been altered. Re-loading during this section, however, may effect the execution of some of the examples and should be avoided.

You're probably getting fed up with sprites 2 and 6 so now let's use some different sprites. We'll begin by clearing the screen and PUTting a sprite near the middle of the top screen row - type:

```
|SCLS:|SET,0:|COL,60:|ROW,8:|SPN,3:|PTBL
```

Note that, although we used PTBL to 'PUT' the sprite onto the screen, there are 6 different ways of 'PUTting' a sprite onto the screen but these will be dealt with in a later section. For now, we'll run through the sprite operations in a similar order to that of the previous section - type:

```
|COL,68:|INVS:|PTBL
```

We have inverted the sprite and PUT it next to the original. Unless we alter the sprite again, subsequent PUT's will always produce the inverted sprite - type:

```
|COL,56:|PTBL
```

The sprite is still inverted but we can easily restore it to its former glory by typing:

```
|INVS
```

To prove that it has been altered in memory let's have another look at it - type:

```
|PTBL
```

Now let's use the sprite scrolls to produce a smooth diagonal scroll - type:

```
|NPX,1:FOR I%=1 TO 512:|WSVN:|WSL1:FOR J%=
40 TO 64 STEP 8:|COL,J%:|PTBL:NEXT J%:NEXT I%
```

This example combines a vertical and horizontal scroll to produce a diagonal scroll and then repeatedly PUT's the sprite at 4 adjacent screen positions. A similar effect could have been achieved by scrolling four adjacent screen windows but the result would have been jagged movement.

We can produce a kaleidoscope effect again, this time using a sprite. Type in the following short program and RUN it:

```
5 ' KALEIDOSCOPE
10 ¦SCLS:¦CLLO:¦COL,20:¦SPN,1:¦PTBL
20 ¦NPX,1:¦COL,40
30 FOR I=1 TO 256
40  ¦ROW,8 :¦PTBL:¦MIRS
50  ¦COL,44:¦PTBL:¦FIPS
60  ¦ROW,24:¦PTBL:¦MIRS
70  ¦COL,40:¦PTBL:¦FIPS
80  ¦WSVN:¦WSL1
90 NEXT I
```

Line 10 clears the screen and 'PUTS' an original copy of the spider sprite to the left of the kaleidoscope area.
Line 20 NPX is set to +1, i.e upward 1 pixel scroll and COL is set to 40.
Line 30 The LOOP count is set to 256. This will ensure that, if break is not pressed, the sprite will finish in its original form, for later use.
Line 40 'PUTs' the first sprite and mirrors it about its vertical centre.
Line 50 Places the sprite in the next position and mirrors it about its horizontal centre.
Line 60 'PUTs' the sprite further down the screen and performs the same operation as line 40.
Line 70 As line 50 but places the sprite in the bottom left hand corner.
Line 80 Scrolls the sprite diagonally with wrap.
Line 90 Loops back to line 30.

Before concluding this section it is worth briefly mentioning that sprites can be rotated and FILLed by 'PUTting' them onto the screen,carrying out the operation and then 'GETting' the transformed image back into the same or another sprite. This is left as an exercise for the user. Remember you will need a border around the sprite before FILLing or the INK may leak over the whole screen!

Let's move on now to the third group of operations which also deal with sprites but are slightly more flexible, albeit cumbersome to use.

SPRITE WINDOW OPERATIONS

The facilities in this group are the same as those available in the previous group with the exception of Y-expansion, and vertical mirroring. The syntactic difference between this and the former groups is that commands in this group are suffixed with 'P' for 'part of sprite' instead of 'S' for 'sprite' or 'V' for 'video'. To define a sprite window however requires 5 parameters and this group uses the 5 variables SPN,COL,ROW,LEN and HGT. SPN identifies the sprite, COL and ROW locate the window and HGT and LEN give the window its dimensions. The operations generally operate noticeably slower than their previous equivalents.

Since the nature of the operations themselves has already been covered, we'll restrict ourselves to a few examples which illustrate the use of sprite windows. Again we'll start by clearing the screen and setting up the sprite window ready for the operation - type:

```
|SET,0:|SPN,3:|COL,2:|ROW,8:|LEN,4:|HGT,16
|SET,2:|SPN,3:|COL,40:|ROW,8
```

Remember that we are not going to see the result of our handy work until we display the sprite. Type:

```
FOR I%=1 TO 512:|SET,0:|WPL1:|SET,1:|PTBL:NEXT I%
```

This will scroll a window in the centre of sprite 3 one pixel left. The sprite is repeatedly 'PUT' so that the changes can be seen.

The other operations work in exactly the same way so there is no point in over-indulging here. Try some experimentation for yourself and be sure you're happy before moving on to the next section.

SPRITE/SCREEN OPERATIONS

This very important group of commands are used to move data between the screen and memory. Without these commands it is not possible to display sprites and for this reason PTBL has already been introduced to illustrate the effect of the previous operations.

There are two directions of movement - from sprite to screen (PUTs) and from screen to sprite (GETs). In each case the data at the source remains unchanged. Each command has three parameters; SPN, COL and ROW. SPN holds the number of the sprite and COL and ROW hold the screen co-ordinates. As before, COL is in bytes and ROW is in pixels. The dimensions of the screen window being moved, or moved into, take their value from the sprite dimensions and if this window lies partially 'off-screen' then it will be adjusted to fit 'on-screen' and only part of the data will move. If a window lies completely 'off-screen' no action will be taken.

All 'PUT' commands (those which move data from a sprite in memory onto the screen) are prefixed with 'PT'. All 'GET' commands (those which move data from the screen into a sprite in memory) are prefixed with 'GT'. There are actually six types of 'GET' and 'PUT' which only differ in the way that their source data interacts with their target data. PTBL, for instance, simply replaces all data at the target with data from the source and all the original data at the target is lost. PTOR, on the other hand, performs a logical 'OR' between the source data and the target data and leaves the result in the target. We will now go through all six operations in detail, with examples where appropriate.

Block Moves - 'BL'

All block move commands are suffixed with 'BL'. The effect of a block move is relatively simple to explain. Whatever data was in the target is completely replaced by whatever was in the source. The target data is lost, and two copies of the source data are then in existence. This means that if we do a PTBL to the screen, then all data in the screen window that provided the target will be lost, regardless of the size of the sprite image. If a sprite has dimensions 16 by 16 but its data only occupies the central 8 by 8, then when the sprite is block put with PTBL the whole 16 by 16 area is written to and in this case would leave a 4 by 4 border all around the outside of the image. To see this happen - type:

```
|SCLS:|SPN,53:|COL,40:|ROW,8:|PTBL:|COL,42:|ROW,16:|SPN,54 :|PTBL
```

Notice how the bird wipes out data in a rectangular area around itself and not just over the area of the bird. Likewise, we can block move data from the screen to a sprite using GTBL - type:

```
|HGT,25:|LEN,5:|INVV:|GTBL
```

What we have done is to invert the screen area holding the original sprite data, then 'GOT' it into the original sprite, replacing the original data with the inverted data. To see this, type:

```
|SCLS:|PTBL:|INVS
```

This clears the screen, PUTs the sprite (which is now inverted) and finally inverts it back again for later use. Of course, it would have been much easier to simply invert the sprite directly, but the above does serve to demonstrate the operation of GTBL.

Logical ORs — 'OR'

All moves with logical OR are suffixed appropriately enough with 'OR'. The effect of PTOR and GTOR are not quite so simple to explain, and if you are not familiar with boolean algebra and binary numbers then a short maths lesson could be very beneficial in the long run. If you are already conversant with the former then skip this next section.

All information in the Amstrad (or any other 8 bit microcomputer for that matter) is stored as a series of 'bytes'. A byte contains 8 'bits' and each bit can contain a 1 or a 0. A byte of sprite or character data in the Amstrad represents 4 pixels in 4 colour mode or 2 pixels in 16 colour mode. This means that a pixel in 4 colour mode contains 2 bits and a pixel in 16 colour mode contains 4 bits. Let's summarise this below; in 4 colour mode:

Pixel State	INK Number
00	0
01	1
10	2
11	3

and in 16 colour mode:

Pixel State	INK Number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

TABLE 1

What happens when two binary numbers are ORed together can be seen in this next example. Supposing we 'OR' together a pixel with INK 10 and a pixel with INK 12.

	1010	INK 10
OR	1100	INK 12
=	1110	INK 14

If a bit is set in the first pixel OR the second pixel, then it is set in the result - so:

1 OR 1 = 1
0 OR 1 = 1
1 OR 0 = 1
0 OR 0 = 0

Now look at the sum above again and see if you know why INK 10 OR INK 12 = INK 14. There is not room to print a whole table here (there are 256 possibilities!) but be sure you can work through the four examples below and get the same results.

```
INK  5 OR INK  8 = INK 13
INK  7 OR INK  9 = INK 15
INK 11 OR INK  4 = INK 15
INK  5 OR INK 13 = INK 13
```

If you're wondering what application all this has to graphics then let's look at some examples - type:

```
        |SCLS:|SPN,54:|COL,40:|ROW,8:|PTBL:|COL,42:|ROW,16:|SPN,53 :|PTOR
```

In fact the only difference between this example and the previous one is that the second sprite has been 'OR'ed into the first instead of 'block PUT'. Notice this time that none of the first sprite has been erased by the blank parts of the second sprite because anything 'OR'ed with 0 remains unchanged. In fact, PTOR and GTOR have fewer applications than the other five options but one fairly useful property is that suitable choices of INK numbers allow data to have display priorities. If, for instance, a sprite with INK colour 1 overlaps two regions with INK numbers 4 and 3 respectively then the sprite will be displayed in the former region with INK number 5 but will not show up in the region with INK 3, thus giving the two regions perspective.

Logical ANDs - 'ND'

All moves with logical AND are suffixed with 'ND'. The effect of PTND and GTND is best explained by further reference to table 1. When two pixels are 'AND'ed together only bits which were set in the source pixel and target pixel will remain set. What happens if we logically 'AND' a pixel containing INK 10 with a pixel containing INK 12?

	1010	INK 10
AND	1100	INK 12
=	1000	INK 8

If a bit is set in the first pixel AND the second pixel then it is set in the result - so

1 AND 1 = 1
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0

Now look at the sum above again and see if you can work out why INK 10 AND INK 12 = INK 8. Below are another four examples, this time of the logical 'AND' operation. Be sure you can agree each result before moving on.

```
        INK  5 AND INK  8 = INK 0
        INK  7 AND INK  9 = INK 1
        INK 11 AND INK  4 = INK 0
        INK  5 AND INK 13 = INK 5
```

Let's look at another example before discussing its application to games writing - type:

```
        |SCLS:|SPN,53:|COL,40:|ROW,8:|PTBL:|COL,42:|ROW,16:|SPN,54 :|PTND
```

Now see if you can explain the result by reference to table 1.

The chief application that logical ANDing is put to is collision detection but the combined use of ANDs and ORs can also produce perspective effects as discussed in the section on ORs. Collision detection is a complicated business and is given a dedicated section later in this manual, but for now let's consider a simple case. Suppose we were writing a game where the 'good guys' were designed with the even numbered INKs and the 'bad guys' were designed with odd colour INKs. If we set up a dummy sprite and fill it with INK 1, then GTND an area of the screen with it, then the sprite will be empty if there were no 'good guys' but will contain data if there were some 'bad guys'. In a later section we will discuss how to test a sprite to see if it contains data.

Logical XOR - 'XR'

All moves with logical XOR are suffixed with 'XR'. XOR stands for exclusive-OR and we'll demonstrate its operation with reference to table 1. To describe it as simply as possible, if adjacent bits in the two pixels are the same, i.e. both 1 or both 0 then the result is 0. If adjacent bits in the two pixels are different, i.e. one is 0 and one is 1, then the result is 1.

	1010	INK 10
XOR	1100	INK 12
=	0110	INK 6

If bits are the same the result is 0, if they are different it is 1 - so:

1 XOR 1 = 0
0 XOR 1 = 1
1 XOR 0 = 1
0 XOR 0 = 0

Now look at the sum above again and see if you can work out why INK 10 XOR INK 12 = INK 6. Below are another four examples. Be sure you can agree each result before moving on.

```
INK  5 XOR INK  8 = INK 13
INK  7 XOR INK  9 = INK 14
INK 11 XOR INK  4 = INK 15
INK  5 XOR INK 13 = INK  8
```

To see the XOR function in operation type:

```
|SCLS:|SPN,53:|COL,40:|ROW,8:|PTBL:|COL,42:|ROW,16:|SPN,54 :|PTXR
```

Although the second sprite has scrambled the first sprite you may feel it is somehow more recognisable than the previous two results. Now we come to the most interesting and indeed useful property of the XOR - type:

```
|PTXR
```

If you have followed the above example correctly you should find that the first sprite has been fully restored and there is no sign of a second sprite. This is because XOR, like inversion, is reversible. Now type PTXR again, noting the result. Then type:

```
|SCLS:|PTXR
```

The sprite should be displayed as if 'PUT' with a block put. This is because the screen was.clear and 'XOR'ing with 0 has no effect. Now type:

```
|PTXR
```

You should find the sprite has completely disappeared, this second property (a direct result of the first) means that we can use the XOR operations to recognise patterns. If we put the sprite we're testing for into a dummy sprite and use GTXR, then scan the dummy sprite; a zero result implies an exact match. Any other pattern and the dummy would have contained some data. So the XORs have several applications, the first property of reversibility can be used to non-destructively move sprite images across screen data (see XMOV, XMVJ, XBNC) and the second property can be used for pattern recognition and hence collision detection.

This concludes the section on logical data movements but the Amstrad has two other types of PUT and GET which we consider in the next section.

Putting In Front - 'IF'

The facility to 'PUT' a sprite in front of screen data is usually only associated with hardware sprites. We felt that this facility was so useful that it had to be added to the package. For this operation INK 0 is attributed with a special property - transparency. This means that if we execute a PTIF then the whole sprite will appear exactly as designed but any empty areas (containing INK 0) will allow the screen data below to show through. This can be very simply demonstrated with our previous two example sprites - type:

```
|SCLS:|SPN,53:|COL,40:|ROW,8:|PTBL:|COL,42:|ROW,28:|SPN,54 :|PTIF
```

This time sprite 54 appears, completely unscrambled, with sprite 53 showing clearly through. In fact, if you continued to PTIF other sprites on top, the 'stack' would grow 'out of the screen' indefinately. GTIF works in exactly the same way but this time the screen data takes priority over the sprite data and of course the result is left in the sprite. We can likewise 'PUT' data 'behind' screen data.

Putting Behind - 'BH'

This uses a similar idea to that employed by the 'IF' commands but on this occasion data from the source is placed behind data at the target and is only visible through transparent (INK 0) parts of the target - type:

```
|SCLS:|SPN,53:|COL,40:|ROW,8:|PTBL:|COL,38:|SPN,54
|ROW,4:|PTBH
```

Again repeated use of PTBH will 'stack images', but this time, 'into the screen'. GTBH works in the same way as PTBH but this time screen data is 'GOT' behind sprite data and the result is left in the sprite.

To summarise, we have considered 12 new commands and these are:

PTBL, PTOR, PTND, PTXR, PTIF, PTBH, GTBL, GTOR, GTND, GTXR, GTIF, GTBH

In fact each of these 12 commands has an optional parameter which we will now discuss.

Collision Counting with GETs and PUTs

It would be useful to know, when we are moving sprite images around, whether or not the image at the source will collide with the image at the target, regardless of the operation to be carried out. Laser BASIC provides this facility but it is kept as an option because its inclusion will slow down command execution somewhat. A collision means that a pixel which does not have an INK value of 0 (i.e. is not transparent) collides with a pixel which likewise does not have an INK value of 0.

Its implementation is fairly straightforward to use. If you wish to execute a PUT or a GET, with collision detection, then the command is followed by a single parameter which is the address of an integer BASIC variable (has a suffix of '%' or is contained in a DEFINT statement). If a collision occurs the BASIC variable is incremented by one, and if it does not, its original value remains. Let's demonstrate the use with a short BASIC program. 'NEW' any current program then type in:

```
5 ' COLLISION COUNTING
10 X%=0:Y%=0:¦SCLS:¦SPN,2
20 FOR I=1 TO 256
30   ¦COL,RND*76:¦ROW,RND*180:¦PTXR,@X%
40   IF X%<>Y% THEN ¦PTXR:Y%=X%
50 NEXT I
```

Line 10	Declares 2 BASIC integer variables, clears the screen and specifies the sprite we are going to use.
Line 20	Sets up the main loop.
Line 30	Selects random ROW and COL values and then XORs the sprite onto the screen; if anything is underneath then X% is incremented by one.
Line 40	Checks to see if X% was incremented by comparing it to Y%. If they differ then a collision was detected and the sprite is PTXRed for the second time, therefore clearing itself out. Y% is then made equal to X% again ready for the next loop.
Line 50	Loops back to line 20.

Print the value of X% to see how many collisions occurred. Note that if X% were greater than 32767 then a negative number would be displayed. In fact, if exactly 65536 collisions occurred then X% will hold 0. Let's now go on to consider the two other types of 'GETs' and 'PUTs'.

SPRITE WINDOW/SCREEN OPERATIONS

The 12 commands in this group are analogous to the 12 commands previously discussed, but instead of the movement being between a whole sprite and the screen, the movement is between a sprite window and the screen. The window is defined by the variables SPN, SCL, SRW, HGT and LEN in exactly the same way as it was for the sprite window operations in the previous chapter. Collision detection and the 6 types of movement are fully supported. Due to the close similarity between these and sprite/screen operations, only one example is given herewith. Enter the following program (NEW any previously entered programs first) then type RUN.

```
5 ' SPRITE WINDOW/SCREEN OPERATIONS
10 ¦SCLS:FOR I%=1 TO 4 : PRINT "ABCDEFGHIJKLMN":NEXT I%
20 ¦LEN,5 :¦HGT,32:¦SPN,6:¦ROW,0:¦SRW,0
30 ¦COL,0 :¦SCL,0 :¦PWBL
40 ¦COL,5 :¦SCL,5 :¦PWOR
50 ¦COL,10:¦SCL,10:¦PWND
60 ¦COL,15:¦SCL,15:¦PWXR
70 ¦COL,20:¦SCL,20:¦PWIF
80 ¦COL,25:¦SCL,25:¦PWBH
90 LOCATE 1,10
```

Line 10	Clears the screen and prints an area of text the size of the sprite we are using.
Line 20	The height and length of sprite 6 are set up and its row and sprite-row are set to 0.
Line 30	Puts the first sixth of the sprite onto the screen using PTBL.
Lines 40 to 80 Continue to put the remaining five sixths of the sprite using different operations.
Line 90	Moves the text cursor down.

This concludes the section covering sprite window/screen operations and leaves us only to consider one other type of movement, sprite to sprite window movement.

SPRITE WINDOW/SPRITE OPERATIONS

Again the same 6 operations are covered by this third group of GETs and PUTs as were covered by the previous two groups. Collision detection is also implemented in exactly the same way. This time, however, data is moved between a whole sprite (whose number is held in SP2) and a sprite window contained in a second sprite whose number is held in SP1. The dimensions of the data block being moved are those of the whole sprite (sprite SP2) and the sprite window is located in sprite SP1 by the usual sprite window variables SCL and SRW. All commands in this group are prefixed with 'PM' or 'GM' as opposed to 'PW' and 'GW' or 'PT' and 'GT'. Again, only one example is really required because of the great similarity between this and the previous two groups. 'NEW' any existing program, enter the following lines and type RUN.

```
5 ' SPRITE/SPRITE WINDOW OPERATIONS
10  |SCLS :|SPN,4 :|HGT,16:|LEN,4:|CSPR:|SPN,5:|CSPR
20  |SPN,6:|COL,26:|ROW,84:|PTBL:FOR T=1 TO 100:NEXT T
30  FOR L=1 TO 2
40   FOR N=0 TO 6
50     |SRW,0 :|SP1,4 :|SP2,6 :|SCL,N*4:|GMBL:|SRW,16:|SP1,5:|GMBL
60     |SRW,0 :|SP1,5 :|PMBL
70     |SPN,6 :|COL,26:|ROW,84:|PTBL:FOR T=1 TO 100:NEXT T
80     |SRW,16:|SP1,4 :|PMBL
90     |SPN,6 :|COL,26:|ROW,84:|PTBL:FOR T=1 TO 100:NEXT T
100   NEXT N
110 NEXT L
```

Line 10 Clears the screen and creates sprites 5 and 4.
Line 20 Places sprite 6 (the OCEAN logo) on the screen and pauses
Lines 30 and 40 are FOR-NEXT loops.
Line 50 Gets 1/7 of the top half of sprite 6 into sprite 4 and 1/7 of the bottom half into sprite 5 using GMBL.
Line 60 Sprite 5 is now put into sprite 6 where sprite 4's data came from using PMBL.
Line 70 Sprite 6 is reput on the screen.
Line 80 Sprite 4 is put into sprite 6 where sprite 5's data came from using PMBL.
Line 90 Sprite 6 is put on the screen.
Lines 100 and 110 loop around.

SPRITE UTILITIES

Before moving onto some of the more exotic features of Laser BASIC we need to look at a group of commands which deal with the creation, deletion, loading, saving and merging of sprites. We'll begin by briefly outlining the way that sprites are stored. This is not essential information and is provided more for academic interest than anything else. If you wish you can skip this section and move onto the commands themselves.

Sprite Organisation

Sprites are organised in two consecutive blocks of memory. The lower block is a table which contains information about the sprite data itself which is in the high block of memory. There are five pointers associated with sprite storage.

Pointer Application

MBOT Points to the first free byte above BASIC. By subtracting the last byte used by sprites it is possible to calculate the free space available for creating new sprites or scrolling buffer (see FREE).

SPST This points to the first byte of sprite data which is actually the first byte after the end of table data.

STAB This points to the first byte of table data and is actually the lowest memory location used to store sprites.

SPND This points to the first free byte after the end of sprite space and generally holds &7000.

SMAX Holds the highest sprite number available to the user. Increasing the value using ESPR will automatically allocate extra table space.

Table Data

The first four bytes of table data correspond to sprite 0 and are reserved for system use. The next four bytes are the sprite 1 information, the next four are the sprite 2 information and so on up to the last four bytes which hold the information for the last sprite whose number is held in SMAX. Thus the size of the table is determined by SMAX and uses 4x(SMAX+1) bytes. Thus the table size varies from 8 bytes for just one sprite up to 1024 bytes for 255 sprites. Thus if the user wanted 32 sprites the table would use 132 bytes and any attempt to create or destroy a sprite with a number greater than 32 would result in an error.

Each sprite entry in the table occupies 4 bytes. The first two bytes hold the address of the sprite data in the sprite data block. If a sprite has been deleted, or has not yet been created, then these two bytes contain zero. The next two bytes are the width and height of the sprite respectively.

Sprite Data

The sprite data itself is organised serially. That means to say that the top pixel line is followed by the second pixel line and so on to the end of the sprite. When a new sprite is created, the table moves down in memory and the new sprite is inserted below the last sprite created. When a sprite is deleted, all the sprites below it move up to reclaim the space, the table is also moved upward and the pointers in the table are adjusted accordingly.

The Utility Commands Themselves

SSPR,e1,e2

This command sets up sprite space. To begin with the table is cleared to zeros and SPST points to SPND (no sprite data exists). The command has two parameters, both of which can be any valid BASIC expression. The first parameter, e1, tells the system how many sprites to reserve table space for. This can have any value between 1 and 255 and can be altered at a later stage. The second parameter, e2, tells the system where to put sprite space. In fact e2 should contain the value of the first byte above sprite space, which will then build downward. When you load Laser BASIC this value is HEX 7000. This puts sprites directly under the Laser BASIC code and there is never really any need to change this so SSPR will seldom, if ever, be executed unless the user wishes to clear all sprites from memory.

DSPR

This command will delete the sprite whose number is held in SPN. If the sprite does not already exist then the error message "** SPN DOESN'T EXIST **" will be displayed. Memory contracts upwards to recover the deleted bytes. If you wish to store a non-relocatable machine code subroutine in a sprite then it should be contained in one of the first sprites created. If a sprite is deleted then all sprites created chronologically after will be relocated. To delete sprite 7 (if it exists) type:

```
        |SPN,7:|DSPR
```

ESPR

This command is provided so that the table size can be altered at any stage. If the table is being extended (SMAX increased) then the table will grow downward and the sprites held in memory will be unaffected. If, however, the table is being contracted, (SMAX reduced) then all sprites currently created with a sprite number greater than the new SMAX will be deleted. Thus ESPR can be used as a sort of block delete. Use this command with caution! ESPR uses SPN to hold the new SMAX. To alter the value of SMAX to become 100 type:

```
        |SPN,100:|ESPR
```

CSPR

This command creates a sprite in memory. The four bytes of sprite information are entered into the table and the whole table is then moved down in memory to accommodate the newly created sprite. At this stage it contains no meaningful data. A 'GET' command is required to load the sprite with an image. If the sprite we are trying to create already exists, then ** SPN ALREADY EXISTS ** will be displayed. If an attempt is made to create a sprite with a number greater than the current SMAX then ** SPN TOO HIGH ** will be displayed. CSPR uses three parameters; the number of the sprite being created, the width of the sprite in bytes and the height of the sprite in pixels. These are held in SPN, LEN and HGT respectively, so to create sprite 15 with a height of 64 pixels (8 characters) and a width of 16 (8 characters in 4 colour mode, 4 characters in 16 colour mode) we would type:

```
|SPN,15:|HGT,64:|LEN,16:|CSPR
```

In this example our new sprite would use HGTxLEN=64x16=1024 bytes.

IMPORTANT NOTE:

Be sure that every time you adjust the top of BASIC with the MEMORY command that you tell the system you have done so. To do this you must use MSET,e1 where e1 is HIMEM+1. If we were changing the top of BASIC to be 20000 decimal, we would type:

```
MEMORY 20000:|MSET,20001
```

The system has now been informed of the change and the FREE command can be used to check that enough room is actually available to create a new sprite and a simple test to check if our last example (creating a 64x16 sprite) would have worked would have been the following:

```
X%=0:|FREE,@X%:IF X%<64*16 THEN PRINT "NO ROOM"
```

If an attempt is made to create a sprite which is too large then **OUT OF MEMORY** will be displayed and no action taken.

Buffer Space

The free space between the top of BASIC and the bottom of sprites is also used by other Laser BASIC commands and great care should also be exercised to ensure that there is sufficient free space to execute them. Below are the commands which require buffer space together with an indication of how much they need.

WSVN These are the vertical scrolling commands. Each one will require: NPX
WPVN times the width of the area being scrolled; bytes. So to scroll a
WVVN sprite 12 bytes wide, in either direction, by 8 pixels will require
96 bytes. To scroll a screen window 40 bytes wide, by 5 pixels, would require 200 bytes.

We've probably dwelt long enough on memory management so let's move on to consider the rest of the sprite utility commands.

RSPR,e1

This command will relocate sprite space in its entirety. That means to say it will move the table and the sprite data as well as adjusting all entries in the table and the sprite space pointers. In fact this command will very rarely be used because relocating upward from the default position would over-write Laser BASIC's system variables and relocating downward would reduce the amount of free space. It does, however, have some advanced applications which are beyond the scope of this section. It has only one parameter, e1, which tells the system the size and direction of the relocation. A positive value will relocate sprite space to a higher address and a negative value will relocate sprite space to a lower address. Suppose we wished to reserve 256 bytes for a machine code subroutine to be loaded under Laser BASIC at &6EFF, we would use the following:

```
|RSPR,-256
```

PSPR,@V$

This command saves the current sprites onto tape or disk. It uses one parameter only and this is used to pass the filename. The filename is stored in a BASIC string variable and so the '@' must precede it or an error will be generated. The filename itself must be typed in upper case, and must not exceed 8 characters in length and of these, the last three characters must be "SPR". In fact Laser BASIC stores the sprites in three files. The first will contain the system variables STAB, SPST, SPND and SMAX. The filename will be changed by Laser BASIC so that the "SPR" on the end of the filename becomes "SYS". The second file will contain the table and this time the filename is altered so that the last three characters become "TAB". Finally, the sprite data itself is saved and this time the filename used is the filename originally passed to the command, i.e. ending with "SPR". If, for instance, we wished to save the current sprites to tape or disk with filename "TESTSPR", then we would use the following:

```
A$="TESTSPR":|PSPR,@A$
```

This would save three files: "TESTSYS", TESTTAB" and "TESTSPR". These are all, in fact, saved as binary files and do not use any special format.

GSPR,@V$

This command loads the three files saved using the PSPR command. Again it uses only one parameter, the filename, and the same restrictions apply. You can't accidentally load another binary file (unless it also ends in "SPR") because the filename given must end in "SPR" (or "SPR.BAK"). This restriction may seem an annoyance but if you were able to accidentally load another binary file the consequences would be serious. The three files are loaded and the system pointers set to the values loaded in (so the system is organised in exactly the same way as it was when the sprites were saved) which means that sprites start and finish where they did when you saved them. Be careful that you have enough room to load a particular sprite file or **OUT OF MEMORY** will be displayed and your current sprites corrupted. To load the file "TESTSPR" use:

```
        A$="TESTSPR":|GSPR,@A$
```

MSPR,@V$

This command will Merge a named file of sprites with the sprites already held in memory. The sprite space currently held in memory will expand downwards to accommodate the merged sprites. The system variables are loaded and examined first. At this stage the loaded table is compared with the resident table to ensure that two sprites with the same number will not be created. If a sprite exists in both the resident and loaded table then execution terminates with an error ** SPRITE ALREADY EXISTS **. If an attempt is made to Merge a sprite with a number greater than the resident SMAX then the system will report an error **SPN TOO HIGH **. Once it has been established that both tables contain distinct sprites they are physically Merged and the new sprite data itself is appended to the end of the old sprite data with the appropriate adjustment of pointers. There are two things to be carefully noted. Firstly, if the combined sprite files exceed the space available, the top of BASIC will be overwritten and secondly, if a loading error occurs between the tables being merged and the sprites being loaded then you will have to clear sprite space and start all over again, so make sure you have copies of both sprite files safely saved on tape or disk!! If you wished to Merge a file named "TESTSPR" with the sprites currently in memory then you would type:

```
        A$="TESTSPR":|MSPR,@A$
```

Incidentally, the Merge facility can be used to force sprites to load to any address you require. Suppose you wished to load a file of sprites to HEX 6000 and the maximum sprite number in the file to be merged is 32 (it is assumed here that you do not wish to preserve your resident sprites), then type:

```
        |SSPR,32,&6000:A$="TESTSPR":|MSPR,@A$
```

Occasionally you will not be able to merge two sets of sprites because of conflicting allocations of sprite numbers, the next two commands have been provided to circumvent this problem.

RNUM

This command has two parameters which are held in SP1 and SP2. SP1 is set to hold the number of the sprite to be renumbered and SP2 is set to hold the new number that the sprite will be allocated. The old sprite is deleted and the new sprite is created. In fact the operation takes place entirely within the table and sprite data is completely unaffected. Errors will be generated if SP1 does not exist, or if sprite SP2 already exists, or if renumbering would cause a sprite to be created with a number greater than SMAX.

ADNM

This is a particularly useful command when used in conjunction with the Merge facility. It allows an offset to be added to the numbers of all the sprites so far created. In fact the offset can be negative so that previously created sprites can be renumbered in a decreasing sense. SPN is used to hold the offset. Supposing 12 sprites exist and are numbered 5 to 16. To renumber these sprites so that they are numbered 20 to 31, type:

```
        |SPN,15:|ADNM
```

SMAX is unaffected by this command as are all the system pointers. Suppose now that we wish to return the previously renumbered sprite numbers back to their original values, i.e. reduce them by 15. We have a problem because we cannot allocate a negative number to SPN. All we do in fact is to add the 2's complement of 15 which is 256-15. To do this - type:

```
        |SPN,256-15:|ADNM
```

The fact that SPN is treated as a 2's complement number in this case may cause confusion so let's just consider it a little further. In this instance SPN is assumed to hold a value in the range -128 to +127. So if we assign 128 to SPN it is treated as holding -128. What all this boils down to is that the largest positive increment we can add is 127, and if we want to use a negative increment, we subtract it from 256 in the aforeseen manner.

ISPR,@V1,@V2,@V3,@V4

This command is used in various applications to interrogate sprite and system details. Before execution, SPN is loaded with the number of a sprite to be interrogated. If the sprite does not exist, SPN will be set to 0 and no meaningful information will be conveyed. If, however, the sprite whose number is held in SPN is found to exist then the following information will be returned:

The BASIC integer variable V1 will contain STAB
The BASIC integer variable V2 will contain SPST
The BASIC integer variable V3 will contain SPND
The BASIC integer variable V4 will contain the address of the actual serial sprite data.

In addition to this, HGT and LEN will be set to hold the dimensions of the interrogated sprite. Be careful not to forget the '@'s in front of the four BASIC integer variables in the parameter list. If any of the four variables has not been previously declared then an "Improper Argument" error will be generated.

```
5 '  EXAMPLE OF SPRITE INTERROGATION
 10  INPUT;"SPRITE NUMBER";S%:PRINT
20   V1%=0:V2%=0:V3%=0:V4%=0:H%=0:L%=0
30   |SPN,S%:|ISPR,@V1%,@V2%,@V3%,@V4%
40   IF V4%=0 THEN PRINT "SPRITE ";S%;" DOES NOT EXIST":GOTO 110
50   PRINT"STAB    =";V1%
60   PRINT"SPST    =";V2%
70   PRINT"SPND    =";V3%
80   PRINT"ADDRESS=";V4%
90   |HGTQ,@H%:PRINT"HEIGHT =";H%
100  |LENQ,@L%:PRINT"LENGTH =";L%
110 PRINT
120 GOTO 10
```

FREE,@V1

The other sprite utility commands have specialised applications which will be dealt with in later sections, but there is one more command which is of immediate interest in this section - FREE. This is similar to the BASIC equivalent FRE which is a function that returns the amount of free space in the BASIC area. FREE returns the amount of free space between the top of BASIC and the bottom of sprites. We've already discussed the operations which utilise this area for workspace (see CSPR) and it is wise to keep an eye on the value that this command returns. Note that the value given is only accurate if the system variable MBOT has been kept up to date (using MSET) each time the BASIC MEMORY command is used. Again, the BASIC variable used to hold the result must have been declared prior to the execution of this command. To find out how much space is free you could use:

```
|FREE,@X%:PRINT X%
```

This concludes the section covering sprite utilities.

Laser BASIC's Dedicated Variables

You have probably noticed by now that Laser BASIC uses a rather unusual regime for handling its parameters. It uses 17 different variables in all and there are 16 different sets of them. The chief reason for this approach is to reduce unnecessary expression evaluation, which in general takes up more processor time than the command itself. More often than not only a few parameters are altered between executions and there is little point in re-calculating unnecessarily. This method also saves space in the source file and means that the graphics routines themselves can access their data very rapidly.

The full set of variables and their descriptions are given in the section "LASER BASIC EXTENSION" under the heading "PARAMETER RELATED COMMANDS". All variable names comprise three characters and are checked for 'RANGE' errors on each assignment. Many of the variables have multiple applications and this means that range checking is not as tight as it might be. HGT and LEN, for example, are generally used to specify window dimensions, but also double up as increments in the MOVE commands and must therefore be allowed to hold negative numbers on occasions.

We have already seen numerous examples of variable assignments, we'll now look at how to interrogate the variables. Just as there are 17 assignments, there are also 17 interrogations and each one takes the same form - the 'BAR' followed by the three characters of the variable name, suffixed with 'Q' (for 'query') and followed by one parameter, which must be the address of a previously declared BASIC integer variable. To interrogate and print the dimensions of a screen window, you could type:

```
|COLQ,@C%:|ROWQ,@C%:|LENQ,@L%:|HGTQ,@H%:PRINT C%,R%,L%,H%
```

Note that if in the above example, any of the variables C%, R%, L% or H% were undeclared, then an "Improper Argument" error would be generated.

In addition to assigning values to variables and interrogating their values, there are five other related commands which are designed to save space and speed up program execution.

EXXV

Each time a Laser BASIC command is executed it selects the parameters it requires from one of the 16 variable sets. The current SET is dictated by the value in the variable; SET. If you are utilising the Amstrad's EVERY and AFTER commands, then a problem can arise. When the routine you are executing under interrupt is entered, the current value in SET is indeterminate. This means that it must be preserved on entering the routine and restored on exit. This could typically be achieved with the following:

```
1000 |SETQ,@X%:REM PRESERVE CURRENT SET
1010 |SET,Y%........
  .
  .
  .
1100 |SET,X%:REM RESTORE SET
1110  RETURN
```

This involves three commands, each of which performs an evaluation and search of the variable area and which is relatively time consuming. To circumvent this problem, Laser BASIC uses an 'alternative' SET variable. Before proceeding any further, let's see how this works with an example. Type in the following short program:

```
5 ' EXAMPLE OF EXXV
10 ¦SCLS:¦SET,0:¦HGT,64:¦LEN,8 :¦COL,40:¦ROW,8
20 ¦EXXV:¦SET,1:¦HGT,32:¦LEN,16:¦COL,48:¦ROW,8
30 EVERY 20,0 GOSUB 50
40 ¦INVV:GOTO 40
50 ¦EXXV:¦INVV:¦EXXV:RETURN
```

Line 10 Clears the screen and sets up a window using SET 0.
Line 20 EXXV makes SET 0 the alternate set. SET 1 is then made the current set and a window is defined using SET 1.
Line 30 The EVERY GOSUB is set up.
Line 40 This just repeatedly executes an INVV which will invert the window defined in SET 1.
Line 50 This is the subroutine that will execute under interrupt. It exchanges the 'current' and 'alternate' SET variables, inverts the window defined in SET 0, restores the 'current' and 'alternate' SET variables and RETURNS.

EXXV was provided to be used with EVERY and AFTER but can also save time and memory in normal applications, particularly where two SETs are being repeatedly exchanged.

SWPS

Laser BASIC provides 12 different modes of sprite movement and in each case, four frame animation is provided. The way that this is achieved is to use a sequence of four sprites whose numbers are held in SP1, SP2, SP3 and SP4. Every time a Move is made the numbers are rotated, so SP1 becomes SP2, SP2 becomes SP3, SP3 becomes SP4 and SP4 becomes SP1. This means that the cycle is continuously repeated in the order SP1, SP2, SP3, SP4, SP1, SP2 ... and so on. Occasionally it may be required to make the sequence run in reverse, i.e. SP4, SP3, SP2, SP1, SP4, SP3 ... and so on. The SWPS command effectively achieves this by simply exchanging SP2 and SP4. The sequence will run in reverse from this point and can of course be re-reversed at any stage. Incidentally, the 'bounce moves' automatically carry out on SWPS every time a bounce occurs and are described in a later chapter.

ASTV

This command assigns the data in a sprite, to the current variable SET. Not only does this save a lot of space in a BASIC program, but it allows parameters to be set up in a fraction of the time required to do it the 'old fashioned' way. It also effectively increases the number of variable sets and is therefore particularly useful in tracking sprite applications. The command uses only two parameters. The current SET and SPN which is set to hold the number of the sprite containing the variable SET data and the sprite containing the data must be created with a height of 1 and a width of 20, or a range error will occur. The data can be entered into the sprite directly by using ISPR to find the address and then POKEing data from data statements. It is obviously more effective to use the sprite generator program to do this, as this will save source code space in the BASIC program. Most assignments use 8 or more bytes, so to assign 6 parameters would take about 50 bytes and quite a lot of processor time. If the data were pre-stored in a sprite then only 20 bytes would be used and less time taken than would be required for a single assignment. If, for example, we have stored in sprite 32 the parameters we require and want to allocate these to SET 9, then we would use:

```
        |SET,9:|SPN,32:|ASTV
```

AVTS

This command carries out the same operation as the previous ASTV command except that data flows in the opposite direction, i.e. from the variable SET to the sprite. Again this can be used to extend the number of variable SETs and, suitably applied, will compact and speed up BASIC programs. As before there are only two parameters, the current set and the destination sprite number. Again, the sprite must have been created with a height of 1 and a width of 20.

ESAV

This is a variation on the previous two commands and allows the current variable SET to be exchanged with the data in the specified sprite. The same two parameters are used and again the sprite must have a height of 1 and a width of 20.

To summarise, it is well worth gaining familiarity with the use of the EXXV, ASTV, AVTS and ESAV commands because their efficient use can not only speed up your program execution but can also save a great deal of BASIC program space. It may therefore be worth spending some time experimenting with them before moving on to the next section.

MOVE COMMANDS IN DETAIL

Let's move on now to some of the more interesting commands. The commands which we are introducing in this chapter will probably turn out to feature more prominantly in your games than any others so far, so it is very important to master them thoroughly.

We should begin by considering the four ways that Laser BASIC can produce sprite movement, we'll begin with the least sophisticated and then work up to the more complex schemes as we go.

Block Over-write

In this scheme, the sprite is moved by simply over-writing itself. This means that the sprite does not move non-destructively and this method cannot be employed if there is any data in the path of the sprite which shouldn't be destroyed. In situations where a sprite is not constrained to move non-destructively, then this type of movement should always be used. Not only is this method at least twice as fast as any other (frequently as much as ten times faster!), but it produces very smooth flicker-free movement anywhere on the screen with no real need to synchronise with frame-flyback. The most popular commercial games use this method extensively as it is often the case that only the main character (usually joystick controlled) needs to move non-destructively. This method does not effectively support collision detection, but again, most of the applications to which the method is put do not require it.

The only precaution the user must take is to ensure that the sprite has a blank (or appropriately coloured) border around the perimeter, so that it over-writes itself fully and doesn't leave data behind. The border must be at least as big as the increments by which the sprite is moving.

Let's look at an example program. 'NEW' any program in memory and be sure that you have the sample sprites in memory before typing and running this example.

```
5 ' EXAMPLE OF BLOCK OVER-WRITE
10 !SCLS
20 !SPN,8:!COL,40:!ROW,0
30 !HGT,1
40 !LEN,0
50 !SP1,8:!SP2,8:!SP3,8:!SP4,8
60 FOR I=0 TO 183
70  !WMOV
80   FOR W=1 TO 50 :NEXT W
90 NEXT I
```

Lines 10 and 20 Clear the screen and specify a sprite and its starting position.
Lines 30 and 40 Height is set to +1, therefore the sprite will move down by 1 pixel every time WMOV is executed and LEN is set to 0, i.e. no horizontal movement.
Line 50 The variables SP1, SP2, SP3 and SP4 are set up with sprite 8, i.e. no animation.
Line 60 The main loop.
Line 70 The WMOV.
Line 80 Since WMOV is very fast, a delay loop is set up.
Line 90 Loops back to line 60.

Exclusive-OR

This is the most popular method for non-destructively moving sprites around the screen (where the micro in question doesn't have hardware sprites) and provides a very distinctive 'feel' to a game. Those of you who are familiar with the Spectrum will have seen this type of movement in quite a few games, although you may not have realised what you were actually witnessing. If you are familiar with the pitfalls move on to the next section. We'll illustrate the effect in our next example and slow it down for you to see it clearly. First we'll move a small sprite around the screen, synchronised to frame-flyback.

```
5 ' EXAMPLE OF EXCLUSIVE-OR
10 |SCLS
20 |SP1,8:|SP2,8:|SP3,8:|SP4,8:|SPN,1
30 FOR I=1 TO 100
40    |COL,RND*76:  |ROW,RND*184:|PTIF
50 NEXT I
60 |COL,-10:|ROW,50:|HGT,1:|LEN,1
70 |XMOV,512,1
```

Line 10 Clears the screen.
Line 20 Sets up all the sprite variables.
Lines 30 to 50 puts 100 sprites at random positions on the screen.
Line 60 Sets the starting position we are going to XMOV and sets the height and length which are set for diagonal movement.
Line 70 XMOV is executed 512 times, with frame-flyback.

The sprite seems to move 'through' the data, neither behind nor in front. The movement is very smooth and quite pleasing to the eye. Now let's run the same program but this time we'll slow it down and see what happened as we moved 'through' the data.

```
5 ' EXAMPLE OF EXCLUSIVE-OR (SLOW)
10  Q%=0
20  |SCLS
30  |SPN,1:FOR I=1 TO 100
40  |COL,RND*77:|ROW,RND*184:|PTIF:NEXT I
50  LOCATE 7,1:PRINT "  ":LOCATE 8,1:PRINT "  "
60  |COL,0:|ROW,50:|SPN,8:|PTBL
70  |HGT,1:|LEN,1
80   FOR I=1 TO 260
90    |XMOV,1,1
100      FOR W=1 TO 50:NEXT W
110    |COLQ,@Q%:IF Q%=81 THEN |COL,-8:|ROW,50
120  NEXT I
```

Line 10 The integer variable Q% is declared.
Line 20 Clears the screen.
Lines 30 and 40 PUTs sprite 1 at 100 random positions.
Line 50 Clears a space for the sprite.
Lines 60 and 70 Set up starting positions and directions for sprite 8.
Line 80 Starts main loop.
Line 90 Performs 1 XMOV with frame-flyback synchronisation.
Line 100 Sets up a 'sub-loop' that slows things down.
Line 110 Checks to see if sprite 8 has reached the edge of the screen and if so puts it on the other side of the screen.
Line 120 Loops back to 80.

Quite a mess, isn't it? Surprisingly, we perceive this quite differently so long as it happens quickly enough. In fact this method of movement has several drawbacks which we'll try and illustrate with a couple of examples. First of all, let's run the example again and this time we'll go flat out instead of synchronising to frame-flyback.

```
5 ' EXAMPLE OF EXCLUSIVE-OR
6 ' WITHOUT FRAME FLY-BACK
7 ' SYCHRONISATION .
10  Q%=0
20  |SCLS
30  |SPN,1:FOR I=1 TO 100:|COL,RND*77
40  |ROW,RND*184:|PTIF:NEXT I
50  LOCATE 7,1:PRINT "  ":LOCATE 8,1:PRINT "  "
60  |COL,0:|ROW,50:|SPN,8:|PTBL
70  |HGT,1:|LEN,1
80  FOR I=1 TO 260
90    |XMOV
100   |COLQ,@Q%:IF Q%=81 THEN |COL,-8:|ROW,50
110 NEXT I
```

It flickers badly - so what went wrong? The problem with this and the next two methods we're going to introduce is that we're caught between the devil and the deep blue sea. On the one hand, we have to wait for the 'dot' (which produces the picture on the monitor or T.V) to be out of the way so that we can move without being caught and on the other hand we have to do everything fast enough for 'smooth movement' to be perceived. In the last example we didn't wait for the 'dot'. The actual move itself is carried out in two phases, the first is to XOR out the previous frame, and the second is to XOR in the new frame. If the dot arrives between these two operations (and there's a fair chance it will if we run flat out), then the sprite 'disappears' before our very eyes. We can demonstrate this effect more clearly by moving a larger sprite, with frame-flyback synchronisation. Type in and run this example.

```
5 ' EXAMPLE OF MOVING LARGE SPRITE
10 |SCLS
20 |SPN,10:|COL,40:|ROW,200:|SP1,10:|SP2,10:|SP3,10:|SP4,10
30 |PTXR
40 |HGT,-1:|LEN,0
50 |XMOV,512,1
```

This time you will see that we acheive smooth continuous movement over part of the screen, lose part of the sprite in others, and completely lose the sprite in others. So what causes this? We can't slow this one down to show you in slow motion because this effect wouldn't happen in slow motion. What is actually happening is that because we've synchronised to frame-flyback the sprite removal and replacement cycle is synchronised to the 'dot'. The sprite is being removed as the 'dot' arrives and it simply disappears. This restriction has prompted some games designers to:

a) Restrict the movement of large sprites to 'safe' areas of the screen.

b) Ensure that sprites which do traverse the whole screen are relatively small.

c) Pause between moves to ensure the 'dot' arrives and animate to disguise jerkiness.

d) Use a more complex (and usually dedicated) scheme.

Of these, option c) is far and away the most useful to us, since most applications require animation anyway. The following example shows how to produce very acceptable movement using XOR (exclusive - OR) movement.

```
5 ' EXAMPLE OF ACCEPTABLE MOVEMENT
6 ' USING EXCLUSIVE-OR .
10 |SCLS
20 |SPN,12:|COL,72:|ROW,182:|SP1,12:|SP2,12:|SP3,12:|SP4,12
30 |PTXR
40 |HGT,0:|LEN,-1
50 FOR I=1 TO 4
60   |XMOV,80,1
70   |COL,72:|PTXR
80   |LEN,-I
90 NEXT I
```

Line 10 Clears the screen.
Line 20 Specifies sprite 12 as the sprite we are going to XMOV and sets its starting position. Notice that SP1, SP2, SP3 and SP4 have the same value so there is no animation.
Line 30 Places the sprite on the screen using XOR.
Line 40 Sets up the direction of the sprite, moving left slowly.
Line 50 Main loop.
Line 60 XMOVs the sprite across the screen.
Line 70 Resets starting column and 're-PUTs' it.
Line 80 The horizontal velocity of the sprite is assigned to the loop variable I.
Line 90 Loops back to 50.

'In-Front' Moving

Although the exclusive-OR movement described in the previous section does produce quite acceptable movement, those of you who have seen hardware sprite based graphics will appreciate the advantage of true non-destructive movement. Unfortunately the Amstrad does not have hardware sprites but we can simulate some of their characteristics with the 'In-Front' and 'Behind' move types.

There is no need to delve into the workings of this method, suffice it to say that it uses a technique akin to that used by film makers who wish to superimpose 'flying saucers' over London. For this method to work, however, we need to use specially prepared (MASKed) sprites. The sprites must be created with twice the width of the image that we wish to display. Those parts of the image which are to be 'transparent' must use INK 0. The image must be wholly contained in the left hand half of the sprite. Once you have created a sprite in this form then all you need to do is use the MASK command to convert it into a MASKed sprite. If this sprite is displayed using any of the normal PUTs or GETs then the data appears disorganised. Don't worry about this because there is a set of commands which deal exclusively with MASKed sprites. To all intents and purposes a MASKed sprite is treated as being half its physical size by the commands which deal with them. Any attempt to MASK a sprite with an 'odd' width will result in a "CAN'T MASK" error, so only use 'even' width sprites. The image can, of course, have an odd width. Masking a previously MASKed sprite will destroy the data.

This type of movement suffers the same timing constraints as does exclusive-ORing but there is one further restriction on its use, not imposed by exclusive-ORing. Each time a MASKed sprite is placed on the screen, the data that is over-written is simultaneously lifted into the sprite for later replacement. This means that the background mustn't evolve during the movement and restricts this mode of movement to stationary screen data. So the sprite being moved 'In-Front' must not be over-written by any other moving object. In fact this rule applies to movement 'Behind' as well. Below is an example of 'In-Front' movement, let the program run to completion since pressing break may corrupt the sprite. If you wish to re-RUN the program delete line 40 or you will mask a previously masked sprite.

```
5   ' EXAMPLE OF IN-FRONT MOVING
10  |SCLS
20  |SPN,1:FOR I=1 TO 200:|COL,RND*77
30  |ROW,RND*184:|PTIF:NEXT I
40  |SPN,13:|MASK
50  |SP1,13:|SP2,13 :|SP3,13:|SP4,13
60  |COL,72:|ROW,100:|HGT,0 :|LEN,-1
70  |FSWP
80   FOR I=1 TO 3
90     |FMOV,236,1
100  NEXT I
```

Lines 10, 20 and 30 Clears the screen and puts sprite 1 at random positions 200 times.
Line 40 Masks sprite 13.
Line 50 Sets all four frames to the same sprite number, i.e no animation.
Line 60 Sets up starting position and direction.
Line 70 'Front swaps' sprite 13 onto the screen.
Line 80 Loop 3 times.
Line 90 'FMOVs' the sprite the full height of the screen, with frame-flyback synchronisation.
Line 100 Loops back.

'Behind' Movement

This uses the same idea as the 'In-Front' move with the only difference being that the sprite moves behind any other data that it moves over. Again the sprite needs to be MASKed in the manner prescribed in the previous section and the same constraints on movement are encountered. You must be sure that the sprite isn't over-written by any other moving sprites whilst it is itself moving.

The MASKing structure is in fact altered by 'behind' movement and it is necessary to reconstruct the original MASK before a particular sprite that has moved 'behind' can subsequently move 'in-front'. A command is provided to do this - RMSK. The following example moves a sprite around the screen, both behind and in-front of screen data and particular note should be paid to re-masking (using RMSK) between movement behind and movement in-front of screen data.

```
5 '  EXAMPLE OF REMASKING
10   |SCLS
20   |SPN,2:FOR I=1 TO 100:|COL,RND*77
30   |ROW,RND*184:|PTIF:NEXT I
40   |SPN,13
50   |SP1,13:|SP2,13:|SP3,13:|SP4,13
60   |ROW,100:|HGT,0:|LEN,-1
70   FOR N=1 TO 5
```

```
80      |COL,72: |FSWP
90      |FMOV,81,1
100     |COL,72: |BPUT
110     |BMOV,81,1
120     |RMSK
130  NEXT N
```

Line 10 Clears the screen.
Lines 20 and 30 Randomly place sprite 2 on the screen.
Line 40 Sets SPN for remasking.
Line 50 Sets SP1, SP2, SP3 and SP4 to 13. No animation.
Line 60 Sets the value to ROW and the direction of movement in HGT and LEN.
Line 70 is the loop.
Line 80 Sets the start COL value and places the sprite on screen using | FSWP.
Line 90 Moves the sprite In-front of data using | FMOV.
Line 100 Sets the start COL value and places the sprite on screen using | BMOV.
Line 110 Moves the sprite behind data using | BMOV.
Line 120 Remasks the sprite.
Line 130 Loops to line 70.

Now that we've looked at the various methods of movement available with Laser BASIC, let's have a look at some of the different move types in more detail and see how they are actually used in practice. Bear in mind what has been said about the advantages and disadvantages of the various methods and always use the simplest scheme you can. If you look at commercial games you may not be surprised to find that they are frequently designed with all this in mind. A typical platform game, for instance, will use block over-write for almost all of the movement, and the backdrops are carefully designed to accommodate this. A sprite which climbs a ladder needn't use XOR if it's designed with the ladder as part of its image!

The Linear MOVE Commands

These commands support all four methods of movement but we'll begin by looking at WMOV which uses block over-write. The linear move commands use 8 graphics variables and 2 or 3 optional parameters.

WMOV

COL The column to move from, measured in bytes.
ROW The row to move from, measured in pixels.
LEN The increment by which to move horizontally, measured in bytes.
HGT The increment by which to move vertically, measured in pixels.
SP1 The number of the sprite to be moved, in this case block over written.
SP2 The number of the sprite to replace the sprite which is to be moved.
SP3 The number of the sprite which will replace SP2.
SP4 The number of the sprite which will replace SP3.

There are a few points to note:

1. Positive values for LEN will cause movement to the right and negative values will cause movement to the left.

2. Positive values for HGT will cause movement toward the bottom of the screen and negative values will cause movement toward the top of the screen.

3. SP1, SP2, SP3 and SP4 hold the sprite numbers of the four frames of the animation. They can hold any numbers you wish and do not need to run sequentially. They could all hold the same number if the sprite were not required to animate.

4. When WMOV, or any of the other MOVE commands for that matter, is executed, it is assumed that SP1 has been previously PUT to the screen, in the case of WMOV it is not necessary for SP1 to be on screen but for the other three commands in the group (XMOV, BMOV, FMOV) it is necessary. The first sprite to be placed on screen by WMOV will be SP2.

5. Because WMOV used block over-write, the collision detection option is not meaningful since a collision will usually be detected with the previous frame that is being over-written.

6. Remember that when a sprite is 'WMOVed' it requires a border around it which is at least as big as the increments of movement or the sprite will leave a trail behind it.

The first example shows a sprite being 'WMOVed' without a border - note the trail left behind.

```
 5 ' EXAMPLE OF BLOCK OVER-WRITE MOVE
 10 |SCLS:|COL,0:|ROW,100
 20 |SPN,14:|SP1,14:|SP2,14:|SP3,14:|SP4,14:|LEN,1:|HGT,0
 30 |WMOV,256,1
```

In this second example we create a larger sprite and place this smaller one (which we wish to WMOV) inside it - the trail is now removed.

```
5 ' EXAMPLE OF BLOCK OVER-WRITE MOVE II
10 |SCLS
20 |COL,2:|ROW,8:|SPN,14:|PTBL:|SPN,15:|DSPR:|HGT,18:|LEN,12:|COL,0:|ROW,6:|SPN,
15:|CSPR:|GTBL
30 |SPN,15:|SP1,15:|SP2,15:|SP3,15:|SP4,15
40 |LEN,1:|HGT,0
50 |WMOV,512,1
```

Line 10 Clears the screen.
Line 20 Puts sprite 14 on the screen then creates sprite 15 (slightly larger than 14), then gets the screen image of sprite 14 into sprite 15.
Line 30 Specifies movement variables.
Line 40 Sets up the direction the sprite will move in.
Line 50 WMOV sprite 15, 512 times with frame-flyback synchronisation.

The WMOV command, like all other MOVE commands, can be executed in a machine code loop, with or without collision detection and frame-flyback synchronisation. To execute WMOV in a machine code loop we merely follow the command with two parameters. The first is the number of times we wish the command to execute and the second tells the system whether to synchronise with frame-flyback or not. In fact block over-write operations do not really need to synchronise with frame-flyback because they don't suffer from the same flicker problems as other methods. In this next example we illustrate movement with and without frame-flyback synchronisation and also illustrate animation.

```
 5 ' EXAMPLE OF BLOCK OVER-WRITE
 6 ' MOVEMENT WITH ANIMATION
 10 |SCLS
 20 |SPN,17:|LEN,0 :|HGT,1:|SP1,17:|SP2,18:|SP3,19:|SP4,20:|COL,40
 30 |ROW,0 :FOR L=1 TO 200
 40   |WMOV,1,1
 50   FOR I=1 TO 60:NEXT I
 60 NEXT L
 70 |ROW,0 :FOR L=1 TO 200
 80   |WMOV
 90   FOR I=1 TO 60:NEXT I
 100 NEXT L
```

Line 10 Clear the screen and set up starting positions.
Line 20 Sets up sprite number, direction and order of animation using SP1, SP2, SP3 and SP4.
Line 30 Sets up first main loop.
Line 40 WMOV with frame-flyback.
Line 50 A small wait loop to stop the sprite animating too fast to see.
Line 60 Loops back to 30.
Line 70 Sets up second main loop.
Line 80 WMOV without frame-flyback.
Line 90 Another small pause loop.
Line 100 Loops back to 70.

XMOV

XMOV uses the same 8 graphics variables as WMOV (so do FMOV and BMOV) and like all the MOVE commands can be followed by up to three parameters. Again there are a few points worth noting.

1. XMOV assumes that sprite SP1 has aleady been XORed onto the screen at the current COL and ROW positions. Failure to do this will leave a copy of SP1 at that position.

2. There is in fact no need to XOR SP1 if movement is to begin from an off-screen position.

3. Collision detection can be gainfully used by XMOV.

4. If XMOV is being repeatedly executed with frame-flyback synchronisation, the upper section of the screen may not permit flicker-free movement for large sprites. In practice, the larger the sprite, the deeper the 'no-go' band.

In this first example we XMOV a sprite from the centre of the screen to the bottom left (and wrap around) without first PUTting SP1 - note that a copy of SP1 is left behind.

```
5 ' EXAMPLE OF XMOV
10 |SCLS
20 |COL,40:|ROW,100
30 |SP1,2 :|SP2,2   :|SP3,2:|SP4,2
40 |HGT,-2:|LEN,1
50 |XMOV,218,1
```

Line 10 Clears the screen.
Line 20 Sets up starting position.
Line 30 Sets up sprite numbers (no animation).
Line 40 Sets up direction.
Line 50 XMOV sprite 2, 218 times with frame-flyback synchronisation.

We now repeat the example but this time initialise the XMOV by using PTXR to PUT sprite SP1 (actually held in SPN) at the centre before moving.

```
5 ' EXAMPLE OF XMOV II
10 |SCLS
20 |COL,40:|ROW,100
30 |SPN,2 :|PTXR
40 |SP1,2 :|SP2,2:|SP3,2:|SP4,2
50 |HGT,-2:|LEN,1
60 |XMOV,218,1
```

Before moving on to our next example, it's worth briefly mentioning the screen 'wrap-around' employed by all linear move commands. The screen is 80 bytes wide and 200 pixels high but sprites can be thought of as moving in a space which is 256 bytes (3 and a bit screens) wide and 256 pixels high.

If a sprite's path crosses an edge of the 'virtual screen', it will wrap around until it eventually arrives on the real screen and can be seen again. This 'wrapping' occurs in any direction.

If we start a sprite 'XMOVing' from a virtual region we do not need to 'launch' it with a PTXR. This next example demonstrates this point.

```
5 ' EXAMPLE OF XMOV III
10 |SCLS
20 |COL,-8:|ROW,100
30 |SP1,2 :|SP2,2   :|SP3,2:|SP4,2
40 |HGT,0 :|LEN,1
50 |XMOV,218,1
```

Let's turn now to collision detection. In all the examples up to this point we have either moved with no parameters, or with 2 parameters (machine code loops), let's look now at an example which demonstrates collision detection.

```
5 ' EXAMPLE OF COLLISION DETECTION
6 ' WITHOUT A MACHINE CODE LOOP
10 X%=0
20 |SCLS
30 |SPN,1 :FOR I=1 TO 40:|COL,RND*77:|ROW,RND*184:|PTIF:NEXT I
40 |COL,-8:|ROW,100
50 |SP1,2 :|SP2,2   :|SP3,2:|SP4,2
60 |HGT,0 :|LEN,1
70 FOR I=1 TO 512
80    |XMOV,@X%
90    IF X%>3 THEN SOUND 1,40,5:X%=0
100 NEXT I
```

Line 10 Declares X%, which is to be used in collision detection.
Line 20 Clears the screen.
Line 30 Places some data on the screen.
Line 40 Makes the start position of the sprite just to the left of the screen i.e -8.
Line 50 Sets up sprite numbers.
Line 60 Sets direction.
Line 70 Start of main loop.
Line 80 XMOV, with X% as the collision detection counter.
Line 90 Checks if X% is different and if so makes a short beep and resets X% back to 0.
Line 100 Loops back to 70.

Collision detection is provided for by passing the address of a BASIC variable (hence the all important '@' in front of the variable name) so that the move routine can increment the variable if a detection is made. XMOV,@X% would move once, and if a collision occurred, increment X%. XMOV,@X%,500,1 would move 500 times (with frame-flyback synchronisation) and increment X% every time a collision was detected (in this case up to 500 times). It is very important not to forget the '@' in front of the X% or the system may well crash. The BASIC variable used for collision detection must be an integer variable, i.e. it must be followed by a '%' or have been declared to be integer in a DEFINT statement. If the variable being used has not yet been declared then an 'improper argument' error will occur. To overcome this, just put in a statement such as X%=0, before the execution of the command.

FMOV

FMOV is in fact identical to XMOV in every respect except that movement is in-front of screen data rather than by exclusive-ORing. Use of the 'in-front' operation does require a few points to be raised.

1. All the sprites used by this type of move (and the FSWP which initialises the movement) must be MASKed before movement can begin. If any of these sprites has been used in 'behind' movement then they will need to be re-masked with the RMSK command. Executing the RMSK command cannot do any harm so if in doubt remask!

2. FMOV assumes that sprite SP1 has already been placed on the screen using the FSWP command (unless movement is beginning from an off-screen position). If SP1 was not in fact placed there using FSWP, then sprite SP1 will be irrecoverably corrupted. For this reason it is a good idea to keep a copy of all your masked sprites, in other sprites so that you can re-construct corrupted sprites using the 'PM' or 'GM' commands.

3. When you have finished FMOVing a sprite it must be removed from the screen by executing a further FSWP, or must be moved 'off-screen'. Failure to do this may also result in a sprite being corrupted.

In the next example we start with four un-masked sprites that are going to form the four frames of an animated sequence, which are then masked ready for movement. If we accidentally corrupt one of these MASKed sprites then we can repeat the procedure. The following short program demonstrates the full use of the FMOV command.

```
5 ' EXAMPLE OF FMOV
10 |SCLS
20 FOR I=21 TO 24 :|SPN,I:|MASK:NEXT I
30 FOR I=1  TO 50 :|SPN,1:|COL,RND*80:|ROW,RND*200:|PTBL:NEXT I
40 |SPN,21:|SP1,21:|SP2,22:|SP3,23:|SP4,24
50 |COL,40:|ROW,0 :|FSWP
60 |HGT,1 :|LEN,0
70 FOR I=1 TO 480
80 |FMOV,1,1
90 FOR W=1 TO 50:NEXT W
100 NEXT I
```

Line 10 Clears the screen.
Line 20 Masks sprites 21 to 24.
Line 30 Puts data on the screen to show that you are actually moving in-front.
Line 40 Sets up sprite numbers for animation.
Line 50 Sets up starting position and FSWPs.
Line 60 Defines direction and the number of pixels in that direction you are going to move in.
Line 70 Main loop.
Line 80 FMOV once with frame-flyback synchronisation.
Line 90 Sets up a delay loop so you can see the animation.
Line 100 Back to 70.

BMOV

The BMOV command is used in exactly the same way as the FMOV command with only a few notable changes.

1. Whereas FMOV requires the first frame to be initially placed on the screen with the FSWP command and then at the end of the move sequence requires the last frame placed to be removed with another FSWP, BMOV requires two distinct commands to be used. The first frame is placed on screen using BPUT and the final frame is removed using BGET. If the first frame is 'off-screen' then BPUT is not required and if the final frame is 'off-screen' then the BGET is not required.
2. When sprites have been used with BPUT, BMOV or BGET they will need to be re-masked, using RMSK before they can be utilised by FSWP and FMOV.

Joystick/Keyboard MOVE Commands

These commands support all four types of movement, use 9 graphics variables and are executed with up to 3 optional parameters. They are very similar to the linear move commands in most respects, but their movement is governed by a user selectable joystick or key row.

These commands use the same 8 graphics variables as their predecessors but with one extra variable which is necessary to select the required joystick or key row. The extra variable used is KEY and is assigned values, and interrogated in the normal way. Joystick 0 is selected by setting KEY to hold a value in the range 72 to 79 and joystick 1 is selected by setting KEY to hold a value in the range 48 to 55. If you do not have joysticks fitted (or if you're not using them) then you can use keys to manoeuvre your sprite.

Table 2 shows which key combinations correspond to which KEY values. Keys in round brackets correspond to keys in the numeric key pad (if fitted). UP, DOWN, LEFT and RIGHT correspond to the cursor keys.

VALUE IN KEY	UP KEY	DOWN KEY	LEFT KEY	RIGHT KEY
0	UP	RIGHT	DOWN	(9)
8	LEFT	COPY	(7)	(8)
16	CLR	[	ENTER	]
24	↑	-	@	P
32	0	9	O	I
40	8	7	U	Y
48	6	5	R	T
56	4	3	E	W
64	1	2	ESC	Q

TABLE 2

The animation cycle is only advanced if a key is pressed (or the joystick activated). The cycle is advanced by one frame regardless of the increment in HGT and LEN, the number of keys pressed, or the direction of movement. Below are a series of examples which illustrate what can be achieved with these commands. The first example demonstrates movement using the joystick and should be skipped if you do not own a joystick.

```
5 ' EXAMPLE OF MOVING A SPRITE USING
6 ' JOYSTICK 0
10 |SCLS
20 |COL,40:|ROW,100
30 |SPN,17:|SP1,17 :|SP2,18:|SP3,19:|SP4,20
40 |PTXR
50 |HGT,1 :|LEN,1
60 |KEY,72
70 |WMVJ,1,1
80   FOR I=1 TO 50:NEXT I
90 GOTO 60
```

Line 10 Clears the screen.
Line 20 Sets up the COL and ROW position for sprite 17
Line 30 Sets up SPN and SP1 to SP4 (the 4 sprites in the animation sequence).
Line 40 Exclusive-OR's sprite 17 onto the screen.
Line 50 Sets HGT and LEN to 1.
Line 60 Sets KEY to 72.
Line 70 Moves and animates the sprites using WMVJ
Line 80 is a delay loop.
Line 90 Loops to line 60.

The next example which is similar to the one above shows how movement and animation can be controlled from the keyboard (see table 2).

KEY is set to 40 which means the '8' key moves and animates sprites up, the '7' key moves them down, the 'U' key moves them left and the 'Y' key moves them right.

```
5 ' EXAMPLE OF MOVING A SPRITE USING
6 ' THE KEYBOARD
10 !SCLS
20 !COL,40:!ROW,100
30 !SPN,17:!SP1,17 :!SP2,18:!SP3,19:!SP4,20
40 !PTXR
50 !HGT,1 :!LEN,1
60 !KEY,40
70 !WMVJ,1,1
80  FOR I=1 TO 50:NEXT I
90 GOTO 60
```

You may have found that using the 'Y' key to move right and the 'U' key to move left a bit difficult to master as the 'Y' key is on the left of the 'U' key. You can however, reverse all directions specified in table 2 by putting negative values in HGT and LEN. The example below is the same as the one above except HGT and LEN are now set to -1 such that the 'Y' key now moves left, the 'U' key moves right, the '7' key moves up and the '8' key moves down.

```
5 ' EXAMPLE OF MOVING A SPRITE USING
6 ' THE KEYBOARD II
10 !SCLS
20 !COL,40:!ROW,100
30 !SPN,17:!SP1,17 :!SP2,18:!SP3,19:!SP4,20
40 !PTXR
50 !HGT,-2 :!LEN,-1
60 !KEY,40
70 !WMVJ,1,1
80  FOR I=1 TO 50:NEXT I
90 GOTO 60
```

The 'Bouncing' MOVE Commands

It may not be immediately apparent why this particular group of MOVEs have been included in Laser BASIC. As shall be seen, they have wide application in games writing, and platform games in particular. Again, these commands support all the move types associated with the previous two groups and use 9 graphics variables. In fact they use the same 9 variables as were employed by the keyboard/joystick moves but for this application KEY is used to hold the number of a special sprite which contains the bounce window parameters.

Bounce Windows

A bounce window is defined by the four variables COL, ROW, HGT and LEN. The window is set up and stored in a sprite before the execution of the bounce commands themselves. The command which performs this operation is BWST (bounce window SET). The window thus created has its top left defined by COL and ROW, and its dimensions defined by HGT and LEN. Sprites do not bounce 'inside' this window, what actually happens is that their COL and ROW values are constrained to lie within the ranges COL to COL+LEN and ROW to ROW+LEN. So HGT and LEN really define the freedom that the sprite enjoys rather than providing some physical border.

Another point worth noting is that the collision detection feature of the first two move types is replaced by a bounce detection option but is invoked in exactly the same way, i.e. using 1 or 3 following parameters. One final point to note is that each time a bounce occurs the frame sequence is reversed. A simultaneous bounce on two perpendicular edges will cause two reversals and the net result is that the frame sequence continues unaltered.

If the motion begins outside the winoow, tnen the sprite will move linearly until it attempts to pass through any part of the window at which point it is 'captured' and will continue to bounce within the window constraints.

Application to Platform Games

The linear moves will not animate through the frame sequence unless either HGT or LEN is non-zero. In other words they have to move to animate. Using the bounce commands a sprite can be bounced up and down by as little as one pixel. If the alternate frames are offset by 1 pixel, then stationary animation can be achieved as in the next example.

```
5 ' EXAMPLE OF STATIONARY ANIMATION
6 ' USING WBNC
10 |SCLS
20 |SPN,18:|NPX,1:|SSVN :|SPN,20:|SSVN
30 |SPN,15:|DSPR :|HGT,1:|LEN,4 :|CSPR
40 |SPN,15:|HGT,2:|LEN,1:|COL,39:|ROW,100:|BWST
50 |KEY,15:|COL,39:|ROW,100:|HGT,1:|LEN,0:|SP1,19:|SP2,20:|SP3,17:|SP4,18
60 |WBNC,1,1
70 FOR I=1 TO 100:NEXT I
80 GOTO 60
```

Line 10 Clears the screen.
Line 20 Scrolls sprites 18 and 20 up by 1 pixel (remember if you re-run this program those sprites will be scrolled again).
Line 30 Creates a bounce data sprite.
Line 40 Sets up the bounce sprite.
Line 50 Sets up the position of the sprite and values of SP1 to SP4.
Line 60 Executes a single bounce.
Line 70 This is a delay loop.
Line 80 Loops around to line 60.

If we analyse a typical screen from any of the popular platform games then it becomes apparent that a lot of the motion can be produced using vertical or horizontal bouncing. At a later time you could run the Laser BASIC demo. One screen of the demo shows a platform type game in which 7 bounce windows have been set up. Sprites which include a rotating eyeball, a toilet and a lift are seen to bounce very smoothly.

Once you have finished this example session you could break into the demo and new the program by typing "NEW" . Laser BASIC and the sprites will still be in memory. So if you type in the following program all 7 sprites will be set bouncing about the screen.

```
5 ' EXAMPLE OF BOUNCE WINDOWS
10 FOR I=102 TO 108
20  |SET,I-102
30  |SPN,I:|ASTV
40 NEXT I
50 A$="WBNCAWBNCBWBNCDWBNCEWBNCFWBNCG#"
60 |ISET,@A$
70 |IRUN,4
```

Line 10 this is a FOR-NEXT loop that will increment from 102 to 108.
Line 20 Sets up the value of I-102 in the variable SET
Line 30 Uses ASTV to set up the variable set I-102 with values stored in data sprite I.
Line 40 Loops to 10.
Line 50 this is the string that contains the 7 WBNC commands to be run under interrupt.
Line 60 Compiles the string A$ into the background table.
Line 70 Runs the background bounce program at every 4th interrupt.

This concludes the chapter on MOVE commands.

BILD

The BILD command is provided to enable the compact storage of screen backdrops. Although it was included with platform games in mind, it should prove useful in just about any game format. As well as enabling data compression it will produce a backdrop very quickly indeed. Essentially the idea is very simple, the screen information is stored in a 'bit matrix' which is held in a sprite. Each bit that is set corresponds to a sprite being 'PUT' and each bit that is not set corresponds to an empty area with the dimensions of the sprite which would have been 'PUT'. The command uses only five parameters.

COL is used to specify the column at which building is to begin. COL can hold a negative value and in fact this feature can be used to move through the 'map'.

ROW is used to specify the row at which building is to begin and ROW can also hold a negative value.

KEY is used to specify the type of operation that BILD will use to 'PUT' the data to the screen. Four types of operation are supported and these are now summarised:

Value in KEY	Operation
0	Block over-write
1	Exclusive-OR
2	PUT in front of current data
3	PUT behind current data

SPN holds the number of the sprite which contains the bit map. Note that there are no constraints on the dimension of this sprite so enormous backdrops can be stored which are many screens high and wide.

SP1 holds the number of the sprite which will be used to build the backdrop. A '1' in the bit map will cause this sprite to be 'PUT' onto the screen (with one of the operations dictated by the value in KEY) and a '0' will cause a gap to be left with the dimensions of the sprite which would have been 'PUT'. Note that a gap is left rather than a blank area being created. In fact INVV can be used, together with a blank sprite to cause blank areas to be cleared.

In our first example we're going to produce a 'P' using sprite 31 as our building block. The data sprite 30 uses only 8 bytes to produce an 8x8 matrix. Sprite 31 has width 10 bytes (1/8th of the screen width) and a height of 25 pixels (1/8th of the screen height) - so the full screen will be utilised.

```
5 ' EXAMPLE OF BILD
10 |SPN,30:|HGT,8:|LEN,1:|CSPR
20 X%=0:  Y%=0
30 |ISPR,@Y%,@Y%,@Y%,@X%
40 FOR I=X% TO X%+7:READ A:POKE I,A:NEXT I
50 |SCLS
60 |COL,0:|ROW,0:|HGT,200:|LEN,80:|KEY,0:|SPN,30:|SP1,31:|BILD
70 GOTO 70
80 DATA 252,102,102,124,96,96,240,0
```

Line 10 Creates a data sprite for BILD
Line 20 Initialises X% and Y%.
Line 30 Uses ISPR to find the address of the data sprite.
Line 40 Reads a data statement at line 80 and pokes the data into the BILD sprite.
Line 50 Clears the screen.
Line 60 Executes BILD which reads the data from the data sprite and puts sprite 31 onto the screen.
Line 70 is a trapped loop.
Line 80 is the data statement (remember once the BILD data sprite has been created lines 10,20,30,40, and 80 would no longer be required).

In our second example we're going to produce a 'P' again using sprite 30 as our data sprite but this time we're going to invert sprite 30 and use sprite 32 (an empty sprite) to ensure that the gaps are cleared of any garbage. We could of course have used SCLS or CLSV before building but we're not always going to build over the whole screen and we often wish to leave areas of the screen unaffected.

```
5 ' EXAMPLE OF BILD II
10 X%=0:Y%=0
20 |SPN,32:|CLSS
30 |SPN,30:|ISPR,@Y%,@Y%,@Y%,@X%
40 FOR I=X% TO X%+7:READ A:POKE I,A:NEXT I
50 |SPN,30:|INVS
60 |COL,0:|ROW,0:|HGT,200:|LEN,80:|KEY,0:|SPN,30:|SP1,32:|BILD
70 |SPN,30:|INVS
80 |COL,0:|ROW,0:|HGT,200:|LEN,80:|KEY,0:|SPN,30:|SP1,31:|BILD
90 GOTO 90
100 DATA 252,102,102,124,96,96,240,0
```

Line 10 Initialises the variables X% and Y%.
Line 20 Clears building sprite 32 to INK 0 using CLSS
Line 30 Finds the address of sprite 30.
Line 40 POKEs the data in the data statement at line 100 into data sprite 30.
Line 50 Inverts the data sprite so all the space around the P will become the data that will be read by BILD.
Line 60 Uses BILD to put all the spaces (sprite 32) on the screen.
Line 70 Inverts the data sprite (resetting the data).
Line 80 Uses BILD to put the P on the screen.
Line 90 is a trapped loop.
Line 100 is the data statement containing the data of the P.

In our third example we're going to produce a 'Q' but this time, instead of using a second blank sprite we're going to utilise sprite 31 by clearing it and then 'GETting' it back again.

```
5 ' EXAMPLE OF BILD III
10 X%=0:Y%=0
20 ¦SPN,30:¦ISPR,@Y%,@Y%,@Y%,@X%
30 FOR I=X% TO X%+7:READ A:POKE I,A:NEXT I
40 ¦COL,0 :¦ROW,0 :¦HGT,200:¦LEN,80:¦KEY,0 :¦SPN,30:¦SP1,31:¦BILD
50 ¦SPN,31:¦CLSS  :¦SPN,30 :¦INVS
60 ¦COL,0 :¦ROW,0 :¦HGT,200:¦LEN,80:¦KEY,0 :¦SPN,30:¦SP1,31:¦BILD
70 ¦SPN,31:¦COL,10:¦ROW,25 :¦HGT,25:¦LEN,10:¦GTBL
80 GOTO 80
90 DATA 56,108,198,198,218,204,118,0
```

Line 10 Initialises the variables X% and Y%.
Line 20 Finds the address of sprite 30.
Line 30 POKEs the data in the data statement at line 90 into data sprite 30.
Line 40 Uses BILD to put the Q on the screen.
Line 50 Clears the brick sprite (sprite 31) to INK 0 using CLSS and inverts the data sprite.
Line 60 Uses BILD to put the spaces around the Q using the cleared brick sprite.
Line 70 GETs the brick from the screen back into sprite 31.
Line 80 is a trapped loop.
Line 90 is the data statement containing the data of the Q.

Collision Detection and Pattern Recognition

We have already seen how collision detection operates with the GETs, PUTs and MOVEs. One of the drawbacks of this method is that it does slow down execution somewhat. To increase the flexibility of the package we have also included another command which will 'SCAN' and area of screen or sprite for pixel data and increment a BASIC integer variable if any data (pixels not set to INK 0) are encountered. There are 3 commands in the group:

SCNV,@V1 Scans the window defined by COL, ROW, HGT and LEN until data is found. Once data is found, the integer variable V1 is incremented and no further execution takes place. Note that V1 must be integer.

SCNS,@V1 Scans the sprite defined by SPN. Again execution terminates once data has been found and the integer variable V1 is incremented.

SCNP,@V1 Scans the sprite window defined by SPN, ROW, COL, HGT and LEN. Execution terminates once data has been found and the integer variable V1 is incremented.

These commands can be used in their own right but are generally used in conjunction with logical GETs and PUTs and dummy sprites to detect collision with specific objects etc.

In our first example we set a sprite bouncing in an empty window and check the top left of the window to see if the sprite passes through it.

```
5 ' EXAMPLE OF SCNV
10  ¦SCLS
20  ¦COL,9 :¦ROW,48:¦HGT,120:¦LEN,66:¦INVV
30  ¦COL,10:¦ROW,50:¦HGT,115:¦LEN,64:¦INVV
40  X%=0
50  ¦SPN,15:¦DSPR  :¦HGT,1  :¦LEN,4 :¦CSPR
60  ¦COL,10:¦ROW,50:¦HGT,100:¦LEN,60
```

```
70  |BWST
80  |SET,10:|COL,10:|ROW,50:|HGT,16:|LEN,8
90  |SET,2 :|KEY,15:|SP1,8 :|SP2,8 :|SP3,8:|SP4,8:|HGT,2:|LEN,1
100 |SPN,8 :|COL,30:|ROW,50:|PTXR
110 |SET,2 :|XBNC,1,1
120 |SET,10:|SCNV,@X%
130 IF X%>6 THEN GOSUB 150
140 GOTO 110
150 SOUND 1,30,25
160 LOCATE 11,24:PRINT "Sprite is in the corner":FOR A=1 TO 1000:NEXT A
170 LOCATE 11,24:PRINT "                       "
180 X%=0
190 RETURN
```

Line 10 Clears the screen.
Line 20 Inverts a window at least 2 pixels larger than the window we are going to use.
Line 30 Defines a smaller window and inverts it, therefore leaving a red border.
Line 40 The collision detection variable X% is defined.
Line 50 Sprite 15 is made into a bounce window data sprite.
Line 60 The dimensions of the window are loaded into the appropriate variables.
Line 70 The above data is PUT into sprite 15.
Line 80 The SET is assigned a number and information. Set 10 will be used to hold the window we are scanning.
Line 90 Set 2 contains the information for the bouncing sprite. Notice the key value being assigned to 15.
Line 100 Since we are 'XBNC'ing we first have to PTXR sprite 8.
Line 110 The actual bounce command with frame-flyback synchronisation.
Line 120 The scanning of the window held in set 10.
Line 130 Tests to see if X% was incremented.
Line 140 Loops back to line 110. This program has to be stopped, using the break key.
Lines 150, 160 and 170 If collision is detected then a short sound is emitted and an appropriate message displayed for a short time before being cleared.
Lines 180and 190 X% is reset back to 0 and a return is made back to the main loop.

Pattern Recognition

Occasionally, the detection of the presence of data is insufficient and the object needs to be identified. Not surprisingly this is a more difficult problem and can be tackled in several ways. If two identical images are XORed together then the result is zero, i.e. all INK 0. In our first example we scan the whole screen for occurences of upper case letter 'A' (note that colour must match as well). When the letter is found it is converted to lower case. Note the use of a dummy sprite to carry out the comparison.

```
5 ' EXAMPLE OF PATTERN RECOGNITION
10  X%=0
20  |SCLS
30  |SPN,12: FOR I=1 TO 50:|COL,RND*70:|ROW,RND*174 :|PTIF:NEXT I
40  FOR I=1 TO 20:LOCATE RND*38+1,RND*23+1:PRINT "A":NEXT I
50  |HGT,8:|LEN,2
60  FOR R=0 TO 200 STEP 8
70   FOR C=0 TO 80 STEP 2
80     |COL,C :|ROW,R
90     |SPN,33:|PTXR:|SPN,34:|GTBL:|SPN,33:|PTXR
100    |SPN,34:|SCNS,@x%
110    IF X%=0 THEN GOSUB 150
120    X%=0
130 NEXT C:NEXT R
140 END
150 |SPN,35:SOUND 1,40,10:|PTBL:X%=0:RETURN
```

Lines 10 to 50 The detection variable is declared and the screen is cleared. Then data is randomly put all over the screen, as are the letter 'A's. Then HGT and LEN are set to the values we are going to use to scan i.e. 1 character.
Lines 55 and 60 The rows are incremented after a whole column has been scanned.
Lines 80 and 90 The COL and ROW are set up and sprite 33 is then exclusively ORed onto the

character we are scanning, thus if the character is an 'A' after XORing it onto the screen we should be left with a blank space, sprite 34 is then filled with this information.
Line 100 Sprite 34 is scanned. Remember if sprite 34 is empty then the data was an 'A'.
Line 110 If no data was detected then GOSUB line 150.
Line 120 X% is reset.
Line 130 The loop goes back to line 60.
Line 140 END.
Line 150 Sprite 35 is a lower case 'a', it is put on the screen and X% is reset and the subroutine returns.

Bear in mind that we do not necessarily need to compare the whole character because occasionally the object we're searching for contains some unique distinguishing feature which we can utilise to save a full scan. Let's repeat the previous example and use a subsection of the 'A' character as the object to be identified. Also bear in mind that this example is made a lot simpler by the fact that 'A' is always displayed on character boundaries.

```
5 ' EXAMPLE OF PATTERN RECOGNITION II
10  X%=0
20  ¦SPN,34:¦DSPR:¦HGT,3:¦LEN,1:¦CSPR
30  ¦SCLS
40  ¦SPN,12: FOR I=1 TO 50:¦COL,RND*70:¦ROW,RND*174:¦PTIF:NEXT I
50  FOR I=1 TO 20:LOCATE RND*38+1,RND*23+1:PRINT "A":NEXT I
60  ¦HGT,8:¦LEN,2
70  FOR R=0 TO 192 STEP 8
80    FOR C=0 TO 78 STEP 2
90    ¦COL,C :¦ROW,R
100   ¦SPN,33:¦SCL,0:¦SRW,0:¦HGT,3:¦LEN,2:¦PWXR:¦SPN,34:¦GWBL:¦SPN,33:¦PWXR
110   ¦SPN,34:¦SCNS,@x%
120   IF X%=0 THEN GOSUB 160
130   X%=0
140 NEXT C:NEXT R
150 END
160 ¦SPN,35:SOUND 1,40,10:¦PTBL:X%=0:RETURN
```

Line 10 X% is declared.
Line 20 Sprite 34 is deleted and created with height 3 pixels and width 1 byte.
Line 30 The screen is cleared and sprites are randomly put over the screen, as are the letter 'A's.
Lines 100 and 110 Instead of getting the whole character into sprites 33 and 34 only the top 3 pixels are used, this speeds up the scanning greatly.

The problem is also complicated when the object we're looking for is 'in-front of' extraneous data. When this is the case we have to mask out all the extraneous data using a logical 'AND' with the object we're testing and then use an XOR to see if the whole object is there. The following example illustrates this.

```
5 ' EXAMPLE OF PATTERN RECOGNITION III
10  ¦SCLS
20  ¦SPN,34:¦DSPR:¦HGT,8:¦LEN,2:¦CSPR:¦CLSS
30  FOR N=1 TO 400 STEP 4:PLOT 1,N:DRAW 650,N,2:NEXT N
40  FOR I=1 TO 20:¦ROW,INT(RND*24)*8:¦COL,INT(RND*40)*2:¦SPN,33:¦PTIF:NEXT I
50  FOR C=0 TO 78 STEP 2
60    FOR R=0 TO 192 STEP 8
70    X%=0
80    ¦COL,C :¦ROW,R :¦INVV :¦INVV
90    ¦SP1,34:¦SP2,33:¦SCL,0:¦SRW,0:¦GMBL:¦SPN,34:¦GTND:¦GMXR:¦SCNS,@X%
100   IF X%=0 THEN GOSUB 130
110 NEXT R:NEXT C
120 END
130 ¦SPN,35:SOUND 1,40,10:¦PTBL:RETURN
```

Line 10 Clears the screen.
Line 20 Creates sprite 34.
Line 30 Draws lines on the screen on which the 'A's will be placed 'in-front'.
Line 40 Puts 20 'A's randomly on the screen.
Lines 50 and 60 are the FOR-NEXT loops to produce a total scan of the screen.
Line 70 Sets X% to 0.
Line 80 Sets up COL and ROW and INVERTS the area of the screen being scanned (so you can see what area is being scanned).

Line 90 Gets the data at COL and ROW into memory using GMBL and then this data is GTND'ed and GTXR'ed. Sprite 34 is scanned for data.

Line 100 Checks to see if any data exists, if none, the A has been recognised and the subroutine is called at line 30.

Line 110 Loops back.

Line 120 ENDs the program.

Line 130 is a subroutine that produces a BEEP and puts a lower case 'a' to signal that the A has been recognised.

Occasionally this test will fail because an object contained all the data we were testing for and more. The next trivial example illustrates this pitfall.

```
5 ' EXAMPLE OF SCNS II
10  |SCLS  :|SPN,34:|CLSS
20  |SPN,12: FOR I=1 TO 50:|COL,RND*70:|ROW,RND*174:|PTIF:NEXT I
30  FOR I=1 TO 10:|ROW,INT(RND*24)*8:|COL,INT(RND*40)*2:|SPN,33:|PTIF:NEXT I
40  FOR N=1 TO 20: LOCATE (RND*39)+1,(RND*20)+1:PRINT CHR$(143):NEXT N
50  FOR C=0 TO 78  STEP 2
60   FOR R=0 TO 192 STEP 8
70     X%=0
80     |COL,C :|ROW,R :|INVV :|INVV
90     |SP1,34:|SP2,33:|SCL,0:|SRW,0:|GMBL:|SPN,34:|GTND:|GMXR:|SCNS,@X%
100    IF X%=0 THEN GOSUB 130
110 NEXT R:NEXT C
120 END
130 |SPN,35:SOUND 1,40,10:|PTBL:RETURN
```

Line 10 Clears the screen.

Line 20 Puts 50 'car' sprites randomly on the screen.

Line 30 Puts 10 'A's on the screen.

Line 40 Puts 20 random yellow blocks on the screen. This block contains all the neccesary data we are testing for. Hence the scan routine is tricked into putting a lower case 'a'.

Lines 50 and 60 are the FOR-NEXT loops that calculate the ROW and COL values.

Line 70 Sets X% to 0.

Line 80 Sets ROW and COL and inverts the part of the screen currently being examined.

Line 90 Scans for an 'A'.

Line 100 If an 'A', or in this example the yellow block, exists then the subroutine at line 130 is called.

Line 110 Loops back.

Line 120 ENDs the program.

Line 130 is the subroutine that registers that the A has been recognised.

Note that if the object we're testing for is 'behind' data we cannot test for it without using some very elaborate scheme dedicated to the specific task.

The pattern recognition methods we have considered so far might be referred to as 'exact' methods. This means to say that if two objects are XORed together with a zero result then they are definitely identical. We're now going to look at some 'approximate' methods which are relatively quick and simple to implement but are indicative rather than conclusive and should therefore be used with a certain amount of trepidation. We'll now introduce three new commands:

SUMV,@V1 This command will produce the arithmetic sum of all the data in the screen window defined by the four variabales COL, ROW, LEN and HGT. The result is left in the BASIC integer variable
V1.

SUMS,@V1 This command will produce the sum of all the data in the sprite whose number is held in SPN and leave the result in the BASIC integer variable V1.

SUMP,@V1 This command will produce the sum of all the data in the sprite window defined by SPN, ROW, COL, HGT and LEN. The result is left in the BASIC integer variable V1.

There are a few points to note concerning the use of this command.

1. The result is to be stored in a 16 bit variable so if it exceeds 65536 it will begin counting again from 0. In other words, the result is MOD (65536). This means that if the true sum were 65538 it would leave 2 as the result. For this reason it is possible for two distinct objects to be indescernable.

2. As well as the 'ambiguity' introduced by 1. a further limitation stems from the fact that re-ordering the data in the object does not affect the sum. For this reason, two objects may appear to be totally dissimilar and produce exactly the same sum. The following transformations can be carried out on an object without affecting the sum of its data:

> Scrolls (with wrap) vertically, scrolls (with wrap) left or right by 1 or 2 whole bytes and vertical mirroring.

It is relatively easy, however, to design distinct objects with differing sums and the chance of two unconnected objects having the same 16 bit sum are remote. More importantly, it is easy to check for an 'ambiguity' and this should be standard practice.

The following examples illustrate the use and some of the pitfalls of these commands:

```
5 ' EXAMPLE OF SUMS AND SUMV
10  |SCLS
20  S%=INT(RND*4)+17
30  |COL,0:|ROW,0:|SPN,S%:|PTBL
40  T%=0
50  |HGT,25:|LEN,8:|SUMV,@T%
60  FOR N=17 TO 20
70     |SPN,N:|SUMS,@S%
80       IF S%=T% THEN |COL,10:|PTBL
90  NEXT N
100 LOCATE 10,10
```

Line 10 Clears the screen.
Line 20 Picks a sprite from 4 at random.
Line 30 Puts that sprite at the top left of the screen.
Line 40 Declares the variable T%.
Line 50 Scans a window around the sprite using SUMV and stores the result in T%.
Lines 60 to 90 compare the sum values of sprites 17 to 20 with the value attained in line 50. If a match is found that sprite is placed to the right of the original sprite.
Line 100 Moves the cursor down.

The example below is exactly the same as the one above except the sprite on the screen is FIPVed before being scanned. SUMV is still able to recognise it as a non-FIPVed sprite image.

```
5 ' EXAMPLE OF SUMS AND SUMV II
10  |SCLS
20  S%=INT(RND*4)+17
30  |COL,0:|ROW,0:|SPN,S%:|PTBL
40  T%=0
50  |HGT,25:|LEN,8:|FIPV:|SUMV,@T%
60  FOR N=17 TO 20
70     |SPN,N:|SUMS,@S%
80       IF S%=T% THEN |COL,10:|PTBL
90  NEXT N
100 LOCATE 10,10
```

Quite often we require to test whether an object belongs to a set of possible objects. We can of course do this by comparing the result in V1 returned by the SUM commands with a pre-defined array of candidates, but since this exercise involves executing some relatively slow BASIC and since it is so often required, we have extended the SUM commands to deal with this problem at machine code speeds. As well as speeding up your programs, this approach also compacts it considerably. The syntax is as follows:

```
SUMV,e1,e2, ... en,@V1
SUMS,e1,e2, ... en,@V1
SUMP,e1,e2, ... en,@V1
```

The execution of the command is very similar to the previously described SUM with one parameter, but instead of the result being placed in V1, the list of expressions before @V1 are tested against the result for a match. If no match is found V1 is assigned zero. If a match is found then the position in the list of the first expression to match the result is assigned to V1. The following examples illustrate the use of these commands.

```
5 ' EXAMPLE OF SUMS AND SUMV III
10  ¦SCLS
20  S17%=0:¦SPN,17:¦SUMS,@S17%
30  S18%=0:¦SPN,18:¦SUMS,@S18%
40  S19%=0:¦SPN,19:¦SUMS,@S19%
50  S20%=0:¦SPN,20:¦SUMS,@S20%
60  S%=INT(RND*4)+17
70  ¦COL,0:¦ROW,0:¦SPN,S%:¦PTBL
80  T%=0
90  ¦HGT,25:¦LEN,8:¦SUMV,S17%,S18%,S19%,S20%,@T%
100 ¦COL,10:¦SPN,16+T%:¦PTBL
110 LOCATE 10,10
```

Line 10 Clears the screen.
Lines 20 to 50 Store the sums of sprite data in appropriate variables.
Line 60 Picks a random sprite out of 4 possible choices.
Line 70 Puts that sprite on the screen.
Line 80 Declares the variable T%.
Line 90 Scans the window using SUMV and stores in T% which of the 4 values has been matched (T% will return a 0 if no match is found).
Line 100 Puts the matched sprite (16 + T%) on the screen.

This concludes the section on collision detection and pattern recognition. These are fairly advanced programming techniques and will probably require a fair amount of practice to gain familiarisation. We have given some examples of the uses these commands can be put to but with some thought it should be clear that much more sophisticated schemes can be developed with practice.

High Resolution Movement

There are a number of algorithms available for storing and 'PUTting' sprites onto the screen. The fastest methods suffer from the limitations of 'byte resolution'. This means that the smallest step a sprite can take in the horizontal direction is 1 byte (4 pixels in 4 colour mode and 2 pixels in 16 colour mode). This can be overcome by designing sprites in 'intermediate' orientations and sequentially placing them in between increments of COL. In the following example we set up a copy of sprite 17 in sprite 18, scroll it into an intermediate position and then sequentially 'PUT' it, incrementing COL every 2 'PUT's.

```
5 ' EXAMPLE OF HIRESOLUTION MOVEMENT
10 ¦SCLS
20 ¦SPN,17:¦COL,0:¦ROW,0:¦PTBL
30 ¦SPN,18:¦DSPR:¦HGT,20:¦LEN,8:¦CSPR:¦GTBL:¦SSR1
40 ¦SCLS
50 FOR I=0 TO 80
60 ¦COL,I
70 ¦SPN,17:¦PTBL
80 ¦COL,I:¦SPN,18:¦PTBL
90 NEXT i
```

Line 10 The screen is cleared.
Line 20 Sprite 17 is put on the screen.
Line 30 Sprite 18 becomes a copy of sprite 17 (using GTBL) then sprite 18 is scrolled right by 1 pixel.
Line 40 The screen is cleared again.
Lines 50 to 90 Sprites 17 and 18 are put one after another every loop.

This is all very well but we can't use this technique with the 'MOVE' commands because the X-increment has to be a constant. In fact the same result can be achieved by putting Laser BASIC into hi-resolution mode. The command that does this is CLHI (CLLO returns Laser BASIC to normal resolution mode).

In high resolution mode the screen is treated as being 160 columns wide. The value in COL is divided by two and if there's a remainder (i.e. COL holds an 'odd' as opposed to an 'even' value) then 1 is added to the sprite number. So we can move sprite X across the screen with 1/2 byte resolution (2 pixels in 4 colour mode, 1 pixel in 16 colour mode) by making sprite X+1 contain the same image as sprite X but offset toward the right by 1/2 byte. In fact, there is a command which does this - HRSP.

The only variable used by HRSP is SPN. What HRSP actually does is to create a second sprite with sprite number one greater than that in SPN (an error is given if this sprite number has already been allocated), copies the sprite SPN into sprite SPN+1 and then scrolls sprite SPN+1 to the right by 1/2 a byte. In the following examples we show how to use HRSP, CLHI and CLLO with the MOVE commands.

```
5 ' EXAMPLE OF HRSP
10 ¦SCLS
20 ¦CLHI
30 ¦SPN,18:¦DSPR
40 ¦SPN,17:¦HRSP
50 ¦SPN,17:¦SP1,17:¦SP2,17:¦SP3,17:¦SP4,17:¦HGT,0:¦LEN,1:¦COL,-20:¦ROW,80
60 ¦XMOV,230,1
70 GOTO 60
```

Line 10 Clears the screen.
Line 20 Sets the software into HIRES MODE.
Line 30 Deletes the old sprite 18.
Line 40 Executes HRSP on sprite 17.
Line 50 Sets up the parameters for | XMOV.
Line 60 Exclusive-OR's sprite 17 onto the screen.
Line 70 XMOV's the sprite.
Line 80 Loops to line 70.

It is important to note that only the 'GT', 'PT', 'GW', 'PW' and 'MOVE' commands will be affected by the use of CLHI. Screen windows utilised by all other commands still treat the screen as 80 column wide regardless. The following example illustrates this.

```
5 ' EXAMPLE OF CLHI
10 ¦SCLS
20 ¦CLHI
30 ¦SET,1 :¦SP1,17 :¦SP2,17 :¦SP3,17:¦SP4,17:¦HGT,0:¦LEN,1
40 ¦SPN,17:¦COL,-20:¦ROW,180:¦PTXR
50 FOR X=1 TO 160
60  ¦SET,1:¦XMOV,1,1
70  ¦SET,2:¦COL,RND*160:¦ROW,RND*135:¦HGT,RND*60:¦LEN,RND*30:¦IK1,RND*4:¦STCV
80 NEXT X
90 ¦CLLO
```

Line 10 Clears the screen.
Line 20 Sets the hardware into HIRES MODE.
Line 30 Sets up the parameters for | XMOV.
Line 40 Exclusive-OR's sprite 17 onto the screen.
Line 50 is a loop.
Line 60 XMOVes the sprite accross the screen.
Line 70 Creates a random window of a random INK value using STCV.
Line 80 Loops to line 50.

Background Execution of Laser BASIC Commands

One of the most powerful features of Locomotive BASIC is its ability to execute subroutines under interrupt using the EVERY and AFTER commands. Although these will prove extremely useful in games writing they do suffer from one limitation. BASIC does not execute its subroutine the instant it receives the appropriate interrupt, instead it completes execution of the current BASIC command. This introduces an element of uncertainty as to the whereabouts of the 'dot' which scans the screen 50 times a second and builds up the picture you see on the monitor. This slight randomness can cause flicker. To get around this problem, Laser BASIC has its own interrupt mechanism which will execute the instant the interrupt is received. This type of execution is referred to as background execution because routines running in this way will continue to run whatever else the machine is doing. You can even type in your next program with routines merrily running in background. Background programs are automatically terminated by GSPR, MSPR and PSPR but MUST be terminated before accessing tape or disk from locomotive BASIC. There are 3 commands associated with background execution - ISET, IRUN and IEND.

ISET,@A$

This command tells Laser BASIC which extended commands are to be executed in background and which SETs of variables are to be used by each command. The information is passed in a string and has the following format:

"command,set,command,set....#"

In fact there are no spaces or other delimeters within the string and the sets are represented by the letters 'A' to 'P'. All characters MUST be typed in upper case. If, for example, we wanted to scroll a window defined by SET 0 and move a sprite within the parameters in SET 10, we would use the following:

```
        A$="WVR1AXMOVK#":|ISET,@A$
```

Laser BASIC now knows what will be executed but we still haven't set the program running - to do this we use the IRUN command.

IRUN,e1

This is similar to locomotive BASIC's EVERY command. In this case, however, only one parameter is required, a BASIC expressions which sets the frequency of execution. The expression can have any value from 0 to 65535 and sets the number of interrupts which will be allowed in between successive executions. A value of 0 means that execution will occur on every interrupt, a value of 1 means every other interrupt and so on. There are a few points to note.

1. Laser BASIC acknowledges an interrupt every time frame-flyback occurs (which is every 50th of a second) so the maximum execution rate is 50 times a second (corresponding to IRUN,0).
2. If you select an execution rate of 0 (IRUN,0) and the background routine requires more that a 50th of a second to execute then control will not return from the background routine. Whilst this is often the desired effect, the only way to exit is to press the ESC key.
3. If you select an execution rate of 1 or more then the interval of a 50th of a second or more will always be allowed between subsequent executions regardless of the time taken to execute the background routine.
4. If a Laser BASIC error (displayed within paired asterisks) occurs in either the foreground or the background program then the background program is terminated. Pressing the "ESC" key during the execution of the background program will also cause it to terminate.

IEND

This command will terminate execution of a background program. Neither ISET nor IRUN should be executed while a background program is running.

Background Commands

Not all Laser BASIC commands can be executed in background. None of the commands which require following parameters can be executed in background and those commands which have optional following parameters can only be executed without their optional parameters. The latter does not normally cause any difficulty and collision detection is implemented by using tracking sprites which are described in a later section. Commands which can be executed under interrupt are detailed under 'CLASS OPTIONS'.

SOUND

Laser BASIC has only one command concerned with sound effects and music; this command is PLAY. As we shall see however, there are an additional 20 'instructions' which are used with the PLAY command and correspond to the facilities offered by the machines operating system. Sound is dealt with in more or less the same way as tracking sprites, 'tunes' are stored in sprites. There are 20 control codes, and each code may be followed by one or more 'data' bytes. Without doubt, the best way to get to grips with sound is to use the sound generator program which is included in this package.

The Sound Generator Program

To load and RUN the sound program:

Load Laser BASIC then load "SNDGEN" using:

Tape: Wind the tape to the start of the sound generator and type RUN" (see tape map).

Disk: Insert disk and type RUN"SNDGEN

A menu will appear on the screen. We will now deal with each option in turn.

Note: Ensure the keyboard is set to upper case before RUNning.
(If not press CAPS SHIFT).

Option 1 - ENTER SOUND PROGRAM

To select this option, type "1" followed by ENTER. You will first be prompted with "ENTER TARGET SPRITE". This should be any number in the range 1 to 255 (a range error will occur otherwise). If the sprite already exists then you are given the option to use it. If you elect to use it (by hitting "Y") then you can begin program entry. If you elect not to use it (by hitting "N") then you are returned to the "ENTER TARGET SPRITE" prompt. If the sprite didn't exist then you are given the option "CREATE IT (Y/N)". If you hit "N" then you are returned to the "ENTER TARGET SPRITE" prompt. If, however, you hit "Y", then you are asked to enter the sprite dimensions HGT and LEN. Be generous - if you create a sprite that is much larger than your needs then you can always 'crunch' it at some later stage. If its too small then you are stuck! If you try to create a sprite that requires more memory than is available then "INSUFFICIENT MEMORY" will appear and you are invited to re-enter HGT and LEN. The amount of memory available for the program will then be HGTxLEN-4. The 'crunching' process itself requires memory so don't be 'over-generous'!

Once the target sprite is set up you are ready to begin program entry. You will be prompted with "ENTER INSTRUCTION". To enter an instruction, type it in and then hit ENTER. There now follows a list of legal instruction and details of their requirements. Before going into the instructions themselves, it is worth mentioning that the mnemonics for each instruction are a little long winded and once you are familiar with them you may wish to truncate them. The mnemonics are listed in the data statements in lines 1000 to 1040 of the sound generator program. Don't change the order of the instructions, but feel free to edit them and make a customised copy. The LIST tune command will also recognise the new instruction names. Let's now look at the instructions themselves.

"SOUND"

Almost all sound programs will contain the 'SOUND' or 'WAIT-SOUND' instruction. It is normally entered as the first instruction of the program, so that the 'CALL' instructions can be used (to save program space). If the first instruction is not a 'SOUND' or 'WAIT-SOUND' and any of the 'CALL' instructions are executed then "ILLEGAL TRACKER CODE" will be generated at run-time. In fact the 'SOUND' command is not often used (see 'WAIT-SOUND') because if it fails to add a 'SOUND' to the queue then execution continues at the next instruction. If a 'tune' is being entered then this is seldom desirable because the note will be lost. The 'SOUND' and 'WAIT-SOUND' commands will issue the following prompts:

"CHANNEL STATUS" — is first displayed, followed by:

"CHANNEL A (Y/N)" — If you wish the sound to be issued to channel A, type "Y", else type "N".

"CHANNEL B (Y/N)" — If you wish the sound to be issued to channel B as well as, or instead of, channel A, type "Y", else type "N".

"CHANNEL C (Y/N)" — If you wish the sound to be issued to channel C as well as, or instead of, channels A and/or B, type "Y", else type "N".

"RENDEZVOUS WITH A" — If you wish the sound to rendezvous with a sound on channel A, type "Y", else type "N".

"RENDEZVOUS WITH B" — If you wish the sound to rendezvous with a sound on channel B, type "Y", else type "N".

"RENDEZVOUS WITH C" — If you wish the sound to rendezvous with a sound on channel C, type "Y", else type "N".

"HOLD (Y/N)" — If you wish the sound to wait at the head of the queue until it is specifically released, type "Y", else type "N".

FLUSH (Y/N)" — If you wish the sound to flush out all the other sounds in the queue before it type "Y", else type "N".

"ENTER AMPLITUDE
ENVELOPE NUMBER" — You should enter a number in the range 0 to 15 which will select the amplitude envelope to be used by the sound. Typing 0 will cause the volume to remain constant throughout the sound. If you select an envelope number in the range 1 to 15 then you will need to set it up using "AMP-ENV" before executing the sound (or set it up from BASIC remembering that "RUN" will destroy it).

"ENTER TONE
ENVELOPE NUMBER"

This is a number in the range 1 to 15 and selects the tone envelope which should be used. Again the envelope will need to be set up before execution using "TONE-ENV" (or set up from BASIC and avoiding "RUN").

"ENTER TONE PERIOD"

This is a number in the range 0 to 4095 and sets the 'PITCH' of the note (which may be altered by the tone envelope). A high value causes a low note and a low value causes a high note. The tone periods which correspond to actual 'notes' are given in the BASIC manual accompanying your machine.

"ENTER NOISE PERIOD"

This is a number in the range 0 to 31. A value of zero is generally used for 'tunes' and corresponds to 'no-noise'.

"ENTER INITIAL VOLUME"

This is a number in the range 0 to 15 and corresponds to the volume that the note will start to play at (this may be altered by the amplitude envelope). Generally, when an amplitude envelope is employed this will be zero.

"ENTER DURATION"

This is any number in the range -32768 to 32767. If a positive value is entered then it is taken as the absolute duration of the note in 100ths of a second, where a negative value is entered it is taken to be the number of the times that the amplitude envelope should be repeated.

"WAIT-SOUND"

This instruction is identical to "SOUND" in every respect, except that if the queue is found to be full then the program counter is left pointing to the current instruction (which couldn't be executed because the queue was full) and control returns from the PLAY command. The parameters are entered in the manner previously described for "SOUND". "WAIT-SOUND" is almost always used where 'tunes' are concerned.

"RESET"

This instruction has no parameters, so it issues no prompts. The effect of 'RESET' is to clear all sound queues and terminate any sounds currently being executed. It is normally the first instruction to be executed by most tunes.

"RELEASE"

This allows individually 'held' sounds to be RELEASEd. There are three prompts:

"CHANNEL A (Y/N)"

If you wish to release the sound held in channel A type "Y", else type "N".

"CHANNEL B (Y/N)"

If you wish to release the sound held in channel B, as well as, or instead of, channel A, type "Y", else type "N".

"CHANNEL C (Y/N)"

If you wish to release the sound held in channel C, as well as, or instead of, channel A and/or channel B, type "Y", else type "N".

"HOLD"

Hold has no parameters and so there are no prompts. The effect is to halt all sounds immediately without flushing the queues, so that they can be re-started by 'CONTINUE'.

"CONTINUE"

Again, this instruction has no parameters and issues no prompts. Its effect is to re-start all sounds which were frozen by 'HOLD'.

"AMP-ENV"

This instruction has a variable number of parameters (dependant upon the number of sections in the envelope) and envelopes can be hardware or software (see your BASIC manual for full details). The following prompts are issued:

"ENTER AMPLITUDE
This should be a number in the range 1 to 15 and selects
ENVELOPE NUMBER" the envelope which is to be defined.

"NUMBER OF SECTIONS"
This should be a number in the range 1 to 5 and selects the number of section which will need to be entered. Each section will be prompted as follows:

"SOFTWARE ENVELOPE
Typing "Y" will cause the next three prompts, typing (Y/N)"
"N" will cause the two prompts after these, instead.

"ENTER STEP COUNT"
This is a number in the range 1 to 127 (see your BASIC manual).

"ENTER STEP SIZE"
This is a number in the range 127 to -128 (see your BASIC manual).

"ENTER PAUSE TIME"
This is a number in the range 0 to 255 (see your BASIC manual).

or if a hardware envelope was selected:

"ENTER ENVELOPE SHAPE"
This is a number in the range 8 to 15 and selects one of the eight possible hardware envelopes (see your BASIC manual).

"ENTER ENVELOPE PERIOD"
This is a number in the range 0 to 65535 (see your BASIC manual).

"TONE-ENV"

This instruction is similar to "AMP-ENV" in that it has a variable number of parameters (dependant on the number of sections in the envelope) and the following prompts are issued:

"ENTER TONE
ENVELOPE NUMBER"
This should be a number in the range 1 to 15 and selects the envelope to be defined.

"NUMBER OF SECTIONS"
This should be a number in the range -5 to 5 (but not 0) and selects the number of sections which will need to be entered. A negative value indicates a repeating envelope. Each section will be prompted as follows:

"ENTER STEP COUNT"
This is a number in the range 0 to 239 (see your BASIC manual).

"ENTER STEP SIZE"
This is a number in the range 127 to -128 (see your BASIC manual).

"ENTER PAUSE TIME"
This is a number in the range 0 to 255 (see your BASIC manual).

"RE-RUN"

This will cause control to jump to the first instructions in the program (usually a 'SOUND' or 'WAIT-SOUND'). This instruction is seldom used in 'tunes' because the first few instructions are usually only executed once but a sound effect will normally end with a 'STOP' followed by a 'RE-RUN'. 'RE-RUN' has no parameters.

"JUMP"

This instruction is more flexible than 'RE-RUN' because control can be transferred to any part of the program. Only one prompt is issued:

"ENTER PC VALUE"
The number is in the range 1 to 65535. In order to calculate the PC value it is often necessary to enter a dummy value such as 1, then when the whole program has been entered, 'LIST' (option 5) and note the PC value. The true value can then be entered using 'DOKE VALUE' (option 15) or 'OVERWRITE AT PC' (option 2). Tunes are normally terminated by a jump to the first 'CALL' in the program, ready to re-play the 'tune'.

"RE-LIM"

The 'PLAY' command itself is most often executed under interrupt (see ISET, IRUN, IEND). Quite often there are a number of other commands executing under interrupt and if these are graphics commands they will probably be required to be invoked much more frequently than the 'PLAY' command. In order to save processor time the 'PLAY' command has a limit which allows it to execute on selected interrupts. If the limit is 0 it will execute on every invocation, if it is 1 then it will execute on every other invocation, if it is 2 then it will execute on every third invocation and so on. 'RE-LIM' allows you to change this limit from within the program and also to reset the programs own internal counter. If you set 'count' to equal 'limit' then the program will execute on the very next invocation, otherwise it will wait until 'count' equals 'limit' and then execute. After any execution 'count' is automatically re-set to 0. This instruction therefore, has two parameters, entered with one prompt:

"ENTER COUNT,LIMIT" — Both numbers should be within the range 0 to 255 and should be typed with a comma between them before pressing ENTER.

"CALL-CHANNEL"

This instruction enables the user to change the channel status in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program without executing the 'SOUND' or 'WAIT-SOUND'. It is normally used to change the rendezvous requirements before adding a particular sound to the queue. In order to make a tune which uses more than one channel, and hence more than one program (hence more than one sprite!) to start all three channels simultaneously, it is common to set the first sound in each program to rendezvous with the other two. As soon as this first note is executed however, the three harmonies seldom continue to rendezvous and so after the first 'CALL-TONE-PERIOD' or 'CALL-TONE-DURATION' the rendezvous' are normally re-set so that all three channels can play independently. This technique will be apparent in the example tunes, supplied with this package. This instruction has only one data byte but each bit is individually significant so there are 8 prompts, one for each bit. These prompts are discussed in detail under the previously described "SOUND" instruction but are listed below for summary.

CHANNEL A	(Y/N)
CHANNEL B	(Y/N)
CHANNEL C	(Y/N)
RENDEZVOUS WITH A	(Y/N)
RENDEZVOUS WITH B	(Y/N)
RENDEZVOUS WITH C	(Y/N)
HOLD	(Y/N)
FLUSH	(Y/N)

"CALL-AMP-ENV"

This instruction enables the user to change the amplitude envelope number in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program without executing the 'SOUND' or 'WAIT-SOUND' instruction itself. Only one prompt is issued, this is for the envelope number - see 'SOUND'/'WAIT-SOUND'.

"CALL-TONE-ENV"

This instruction is identical to 'CALL-AMP-ENV' except that the tone envelope number is altered - see 'SOUND'/'WAIT-SOUND'.

"CALL-TONE-PERIOD"

This instruction enables the user to change the tone period (which controls the pitch) in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program. This instruction will then execute the 'SOUND' or 'WAIT-SOUND'. If the first instruction is actually 'WAIT-SOUND' as opposed to 'SOUND', and if the queue is full, then the program counter remains pointing at the 'CALL-TONE-PERIOD' instruction and control returns from the 'PLAY' command. In practice, this and 'CALL-TONE-DURATION' are the instructions usually used to add sounds to the queue and 'SOUND' and 'WAIT-SOUND' are rarely executed directly. Only one parameter is prompted for and this is the "TONE-PERIOD" which is a number in the range 0 to 4095.

"CALL-NOISE-PERIOD"

This instruction enables the user to change the noise period in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program without executing the 'SOUND' or 'WAIT-SOUND' instruction itself. Only one prompt is issued - see 'SOUND' or 'WAIT-SOUND'.

"CALL-INITIAL-AMP"

This instruction enables the user to change the initial amplitude in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program without executing the 'SOUND' or 'WAIT-SOUND' instruction itself. Only one prompt is issued and this is for the initial amplitude - see 'SOUND' or 'WAIT-SOUND'.

"CALL-DURATION"

This instruction enables the user to change the duration in the 'SOUND' or 'WAIT-SOUND' data block at the start of the program without executing the 'SOUND' or 'WAIT-SOUND' instruction itself. Only one prompt is issued and this is for the duration - see 'SOUND' or 'WAIT-SOUND'.

"CALL-TONE-DURATION"

This instruction enables the user to change both the tone period and the duration in one instruction, thus saving 1 byte (and some time) when both of these quantities vary simultaneously, which is often the case. As in 'CALL-TONE-PERIOD', the new values are substituted into the data block and then the 'SOUND' or 'WAIT-SOUND' instruction is executed. In the latter case, a full queue will cause the 'PLAY' command to be exited with the program counter still pointing at the 'CALL-TONE-DURATION' instruction. There are two prompts, the first is for the duration and the second is for the tone period -see 'SOUND' or 'WAIT-SOUND'.

"STOP"

This will cause program execution to cease and the 'PLAY' command to be exited with the program counter pointing at the next instruction after the 'STOP'. There are no parameters.

"DONE"

This does not generate any code in the sound program but instead simply terminates program entry and returns to the menu.

"EXIT"

This is the same as 'DONE' except that a byte is placed in the program which tells the 'LIST' command that the end of the program has been reached. Program execution should never reach this point in the program or the system will terminate with "**ILLEGAL TRACKER CODE**"

General Notes:

a) If the sprite you have created to contain the tune is full, then the "END OF SPACE" message will be issued, do not attempt to enter any more data. The data will have been added.

b) If the data you are trying to put into the sprite will not fit, then the "NO ROOM" message will be issued. If this is the case then the data has not been added.

c) If you incorrectly ENTER any of the instructions then the "ILLEGAL INSTRUCTION" message will be issued. All you need to do is re-enter the instruction correctly as no data will have been written to the sprite.

d) Similarly, if you accidentally enter a parameter which is out of range, the "OUT OF RANGE" error message will be issued and you will be prompted to re-enter the data.

e) If you are running other tasks under interrupt, you may find that your tune or sound effect runs more slowly. Only experimentation will establish the adjustment required to the duration but the pitches (set by the tone periods) will be unaffected.

Option 2 - OVERWRITE AT PC

From time to time you may find, when you 'LIST' your program, that you have incorrectly entered the program or missed out an instruction (see Option 4 - INSERT AT PC). If this is so, you can overwrite your mistake with this option. Remember to exit this option with a 'DONE' and not an 'EXIT', which will leave a marker in the middle of your program (unless you're overwriting the last instruction). This facility can also be used to add instructions to the end of a previously entered program. This option will issue the following prompts:

"ENTER TARGET SPRITE" — This is a number in the range 1 to 255 and must be a previously defined sprite. If the sprite does not exist the "NO SUCH SPRITE" message will be displayed and control will return to the menu. If the sprite did exist then the following prompt will continue:

"ENTER PC VALUE" — This should be a value between 1 and 65535, but if it is higher than the sprite size allows then a "NO ROOM" message will be generated as soon as any attempt is made to enter an instruction.

If the PC value is within range then instructions are entered in exactly the same way as they were with Option 1. See 'EXIT' and 'DONE'. In order to establish the PC value at a particular part of the program see Option 5 - LIST.

Option 3 - DELETE AT PC

This option enables the user to delete a chunk of program and move the remaining program back over the recovered memory, thus contracting the whole program by the amount removed. This option will issue three prompts:

"ENTER TARGET SPRITE" — The prompt should be responded to in the same way as the same prompt in the previously described Option 2 - OVERWRITE AT PC.

"ENTER PC VALUE" — Again this prompt should be responded to in the same way as the previously described Option 2 - OVERWRITE AT PC.

"HOW MANY BYTES" — This should be a number in the range 1 to 65535 and is the number of bytes to be deleted from (and including) the selected PC value.

This option may take several seconds to execute and again the relevant PC value is obtained by using Option 5 - LIST.

Option 4 - INSERT AT PC

This option enables the user to make space for a chunk of program and moves the rest of the program (including the instructions at the selected PC) upward in memory to make room. Take care to ensure that sufficient memory is available or the end of your program may be lost. This option issues the same three prompts as Option 3 - DELETE AT PC, and they should be responded to in the same manner. The space allocated is initially filled with zeros.

Option 5 - LIST TUNE

This option enables the user to 'LIST' a previously entered sound program to the screen or printer. An example listing is given at the end of this section. This option issues four prompts:

"PRINTER (Y/N)" — Typing "Y" will send output to the printer only, typing "N" will send output to the screen only.

"ENTER TARGET SPRITE" — The number of the sprite containing the program to be listed (see Option 2).

"ENTER PC VALUE" — The PC value at which to start listing (see Option 2).

"NUMBER OF BYTES" — The length of code to list. If this is longer than the sprite itself then listing will automatically terminate at the end of the sprite. Listing will also terminate if a marker (see 'DONE') is encountered.

Care should be taken to ensure that listing begins at a legal instruction, or the output may be meaningless. If an instruction with an illegal code is encountered then "ILLEGAL INSTRUCTION" is printed and listing continues at the next byte. Listing can be halted and recommenced using "ESC" in the same way as normal BASIC listings. Listings are issued in the following format:

Column 1

This is the absolute address (in HEX) of the current instruction. This will only change if the sprites are deleted, relocated or merged, but bear in mind that it can change and always check if you're unsure.

Column 2

This is the actual byte that is contained in the current address (in decimal).

Column 3

This is the program counter value (in decimal) and it is this number that is used by most options to indicate position in the program.

Column 4

This contains either the current instruction, or the current data which belongs to the current instruction.

Option 6 - SAVE TUNES

This option is provided to enable the user to save the current sprites to tape or disk. Only one prompt is issued:

"ENTER FILENAME TO SAVE UNDER" — The filename should be 8 characters or less in length, and of these, the last three must be "SPR" to indicate a sprite file.

Option 7 - LOAD TUNES

This option is provided to enable the user to load a previously saved file of sprites from tape or disk. If an error occurs during this option (or any other for that matter), simply type "RUN". Only one prompt is issued:

"ENTER FILE TO LOAD" — Again the filename must conform to sprite file format.

NOTE:

To load the demo tunes (which are saved directly after the sound generator program on the tape version) use the filename "MUSICSPR".

Option 8 - MERGE TUNES

This option is provided to enable the user to merge previously saved file of sprites from tape or disk. Again there is only one prompt which asks for the file to merge.

Option 9 - PLAY TUNES

This option enables the user to hear his handiwork. Since there are 3 channels, there are up to three programs which may run together. Often, the rendezvous requirements mean that all three tunes must be played together. This option will play the tunes under interrupt. There are a number of prompts:

"HOW MANY TUNES" — This is a number in the range 1 to 3 and specifies how many tunes are played together, in the case of all 3 example programs, the tune is held in three sprites, one for each channel. None of these will play independantly because of the rendezvous requirements. For each of the sprites there are 3 prompts:

"ENTER SPRITE NUMBER" — This is a number in the range 1 to 255 and is the number of the sprite holding the program we wish to play.

"ENTER PC TO START AT" — This is the program counter value at which execution should start for the particular sprite.

"ENTER LIMIT" — This controls the frequency with which execution is attempted as previously described.

Incidentally, the lines of BASIC which set the tune running may be useful in your own program and are at 2960 to 2990.

Playing the Examples

There are 3 example tunes which are recorded on the tape directly after the sound generator program or saved on disk under the filename "MUSICSPR". Use option 7 to load the example tunes.

Tune 1 This is the music which accompanies the "CHASE" screen in the Laser BASIC demo. It plays on all three channels and the data is stored in sprites 90,91 and 92. All three are initialised with a "PC to start at" of 11 and a "Limit" of 1.

Tune 2 This is the music which accompanies the "platform game screen" in the Laser BASIC demo. It also plays on all three channels but uses sprites 60,61 and 62. Again all three are initialised with a "PC to start at" of 11 and a "Limit" of 5 or less.

Tune 3 This is the music which accompanies the "hunchback" screen in the Laser BASIC demo. Again all three channels are used and this time sprites 93,94 and 95 contain the data. As with tunes 1 and 2 the "PC to start at" is 11 and the "Limit" 5 or less.

Option 10 - STOP TUNE

This simply stops the tune or sound effect from running under interrupt and returns to the main menu. No prompts are issued. Tunes can also be halted by pressing "ESC".

Option 11 - ALTER SPRITE MAX

The sound generator program does not alter the maximum sprite number when it is 'RUN' and all sprites currently in memory are preserved. If, however, you wish to reset the maximum sprite number then option 11 will do this. Only one prompt is issued which asks for the new sprite max. Remember that any currently existing sprites with numbers greater than the new sprite max will be lost.

Option 12 - CRUNCH TUNE

This can be a very useful facility but should be used with caution because of its irreversible nature. As was previously mentioned, it is advisable to work with sprites that are a lot larger than the anticipated requirement because mistakes can easily be made when estimating the exact requirement. The only way to transfer data out of a sprite that has been created too small is to break out of the program and do it by hand, i.e. find the starting address using ISPR, create a new sprite and then transfer using PEEKs and POKEs - not recommended. The sound generator program leaves ample memory for sprites and there should seldom be a shortage of sprite space unless you are merging a file of graphics sprites. Remember that the crunching operation itself will create a new sprite of the required size, before deleting the old one, and so sufficient memory must be available to allow this. There must also be at least one sprite number which has not been used. If insufficient memory is available for the transfer then "INSUFFICIENT ROOM" is reported, if all sprite numbers have been allocated then "NO FREE SPRITE" is reported. If the first free sprite number found is greater than the sprite max then the transfer cannot be completed. The operation issues a number of prompts:

"ENTER TARGET SPRITE" This is a number in the range 1 to 255 and is the number of the sprite to be 'crunched'.

"ENTER PC VALUE" This is the value of the program counter at the last byte used by the sound program. Be careful - the last instruction may be 2 or more bytes long and if you are unsure, add a couple of bytes for safety. In each of the example programs the last instruction is a 'JUMP' (3 byte instruction) so we want the program counter value (column 3 of the listing) at the 'JUMP' instruction itself, + 2. This will point at the last byte.

If the 'crunch' was successful then the new height and width are displayed. Pressing any key will return to the menu. This option can take a fair time to execute.

Option 13 - CLEAR ALL TUNES

This option allows the user to clear all existing sprites from memory and re-set the maximum sprite number. There are two prompts:

"CLEAR SPRITES (Y/N)" — This is provided just in case you have selected option 13 (unlucky for some!) by mistake. If that is the case then just type "N" to return to the menu. To continue with the clearing type "Y" and you will be prompted with:

"ENTER NEW SMAX" — You should now enter the new maximum sprite number which should be a number in the range 1 to 255.

Option 14 - POKE BYTE

This option should be used with great caution and is really an alternative to Option 2 - OVERWRITE AT PC. The 'LIST' option can be used to find the absolute address (column 1) you wish to POKE. Be sure that this address has not changed since you last checked it. There are two prompts:

"ENTER ADDRESS TO POKE" — This will be a number in the range 1 to 7000 HEX. Remember that the address given in column 1 of the listing is a HEX address and therefore requires an ampersand ("&") to be put in front of it, if it is to be entered as a HEX number.

"ENTER BYTE" — This is the value you wish to POKE into the address. It is always a good idea to list the program again after carrying out this option to be sure you have not made an error. The byte must be in the range 0 to 255.

Option 15 - DOKE VALUE

This odd sounding option is the 16 bit equivalent of 'POKE BYTE'. There are two prompts:

"ENTER ADDRESS TO DOKE" — Again this is the address to modify. The least significant byte will go into this location and the most significant byte will go into the next location.

"ENTER VALUE" — This is the 16 bit value which will be DOKEd at the address given.This can be in the range -32768 to 65535.

Option 16 - DELETE TUNE

This option simply allows you to delete an individual sprite and has only two prompts:

"SPRITE TO DELETE" — This should be the number of the sprite to be deleted and should be in the range 1 to 255 (or the maximum sprite number).

"DELETE (Y/N)" — This is provided so that you can abort if you've typed in the wrong sprite number. To abort just hit "N", to proceed and delete type "Y".

Using Sound in your Programs

Once you have created your tunes/sound effects using the sound generator program it is relatively simple to incorporate them in your BASIC program. In fact there is only one command related to the use of sound, but it has two different operations depending on the number of parameters supplied.

PLAY,e1,e2

If the 'PLAY' command has 2 parameters then its action is as follows. The sprite whose number is held in the variable KEY has its program counter set to e1, its count set to zero and its limit set to e2. No instructions within the sound program are executed.

PLAY

If the "PLAY" command has no parameters then its action is as follows. The program within the sprite whose number is held in KEY is executed from its current program counter.

Example

Suppose we have a tune which is contained within 3 sprites numbered 50, 51 and 58. Suppose each one is to be executed with a starting PC of 11 (i.e. there is a 'SOUND' or 'WAIT-SOUND' instruction at the beginning of the program) and a limit of 5. In order to set them running under interrupt using SETs 1, 2 and 4 we would use:

```
10|SET,1:|KEY,50:|PLAY,11,5:A$="PLAYB"
20|SET,2:|KEY,51:|PLAY,11,5:A$=A$+"PLAYC"
30|SET,4:|KEY,58:|PLAY,11,5:A$=A$+"PLAYE#"
40|ISET,@A$:|IRUN,5
```

General Notes on the Use of Sound

1. The 'RUN' command causes a 'RESET' of the sound chip, so if you are playing a tune under interrupt then RUNning a program will have unpredictable results.
2. If you are playing a tune then the first instruction should almost invariably be 'WAIT-SOUND'.
3. If you are playing a tune that uses more than one channel, then it is usually best to write the separate parts of the tune in separate sprites (this avoids continual CALL-CHANNELs).
4. If you are writing a tune in more than one sprite then the first instruction to be executed in the first sprite (as opposed to the first instruction of the program) should be a 'RESET'. This is normally followed by all the envelope definitions and then a 'CALL-CHANNEL' (in each sprite) to ensure that the tunes start together.
5. Where 3 sprites are involved the first note in channel A should rendezvous with B and C, the first note in chanel B should rendezvous with A and C and the first note in channel C should rendezvous with A and B. After the execution of the first 'SOUND' or 'WAIT-SOUND' the rendezvous requirements normally change.
6. If a tune does not start simultaneously on all 3 channels then a dummy envelope with zero volume is normally used to cause the delay whilst still rendezvousing all 3 at the start.
7. Where a tune is to be played repeatedly the last instruction is normally a 'JUMP' back to the 'CALL-CHANNEL' that sets up the rendezvous with the other harmonies. There is no need to re-execute the 'RESET' or envelope definitions.

A typical 3-channel tune may have the following format:

Sprite 1

```
PC  Instruction
1      WAIT-SOUND
11     RESET
12     AMP-ENV
30     TONE-ENV
38     CALL-CHANNEL (set up rendezvous with B and C)
40     CALL-TONE-DURATION (play first note)
45     CALL-CHANNEL (remove rendezvous requirements)
47     CALL-TONE-DURATION (the rest of the tune)
 .
 .
 .
N      JUMP         ('JUMP' back to 'CALL-CHANNEL' at 38)
```

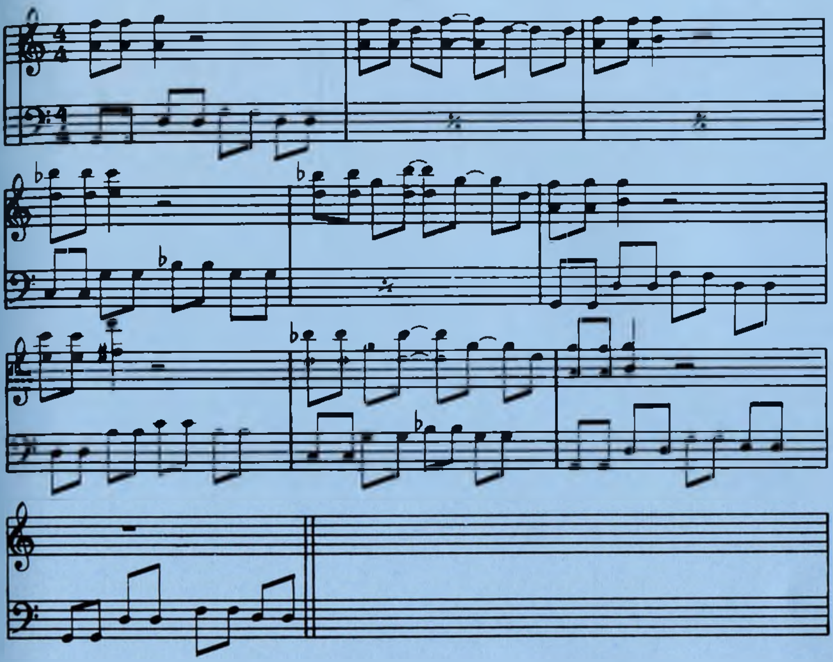
This would be set up with a PLAY,11,LIMIT

Sprites 2 and 3

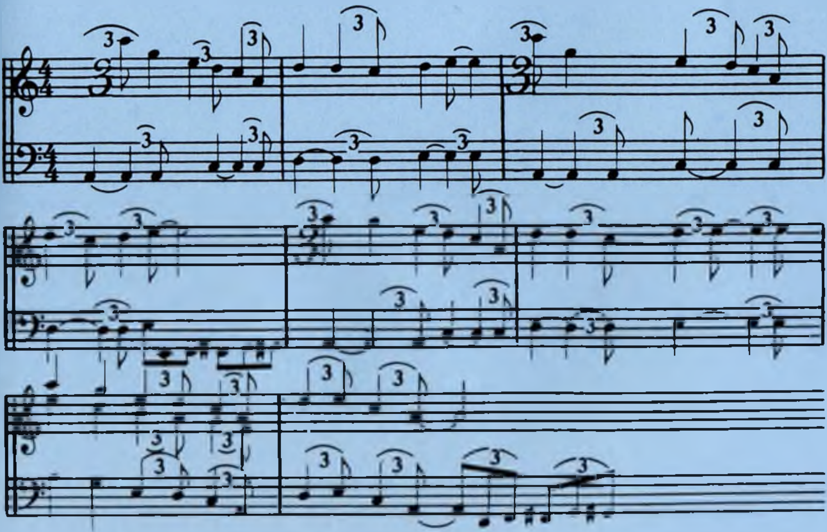
These would take the same form as sprite 1 except that they must not contain a 'RESET'. Again they would be set up to execute from a PC of 11.

If a dummy amplitude envelope of zero volume is to be used to control delayed starts then it would be inserted before the 'CALL-CHANNEL' (shown at a PC of 38 in sprite 1) and would normally be defined in sprite 1. In fact there is no reason why all envelope definitions couldn't be set up in sprite 1.

THE PLATFORM GAME MUSIC



THE HUNCHBACK MUSIC



CHANNEL A

```
6C55 1     1    WAIT-SOUND
                ISSUE ON A
6C57 1     3    AMPLITUDE ENVELOPE= 1
6C58 0     4    TONE ENVELOPE     = 0
6C59 126   5    TONE PERIOD       = 638
6C5B 0     7    NOISE PERIOD      = 0
6C5C 15    8    INITIAL VOLUME    = 15
6C5D 23    9    DURATION          = 23
6C5F 2     11   RESET
6C60 6     12   AMP-ENV
6C61 1     13   AMPLITUDE ENVELOPE= 1
6C62 5     14   NO OF SECTIONS    = 5
6C63 1     15   STEP COUNT        = 1
6C64 0     16   STEP SIZE         = 0
6C65 1     17   PAUSE TIME        = 1
6C66 1     18   STEP COUNT        = 1
6C67 0     19   STEP SIZE         = 0
6C68 10    20   PAUSE TIME        = 10
6C69 1     21   STEP COUNT        = 1
6C6A 241   22   STEP SIZE         = 241
6C6B 1     23   PAUSE TIME        = 1
6C6C 12    24   STEP COUNT        = 12
6C6D 1     25   STEP SIZE         = 1
6C6E 1     26   PAUSE TIME        = 1
6C6F 12    27   STEP COUNT        = 12
6C70 255   28   STEP SIZE         = 255
6C71 1     29   PAUSE TIME        = 1
6C72 6     30   AMP-ENV
6C73 2     31   AMPLITUDE ENVELOPE= 2
6C74 5     32   NO OF SECTIONS    = 5
6C75 1     33   STEP COUNT        = 1
6C76 15    34   STEP SIZE         = 15
6C77 1     35   PAUSE TIME        = 1
6C78 1     36   STEP COUNT        = 1
6C79 0     37   STEP SIZE         = 0
6C7A 10    38   PAUSE TIME        = 10
6C7B 1     39   STEP COUNT        = 1
6C7C 241   40   STEP SIZE         = 241
6C7D 1     41   PAUSE TIME        = 1
6C7E 15    42   STEP COUNT        = 15
6C7F 1     43   STEP SIZE         = 1
6C80 1     44   PAUSE TIME        = 1
6C81 15    45   STEP COUNT        = 15
6C82 255   46   STEP SIZE         = 255
6C83 1     47   PAUSE TIME        = 1
6C84 11    48   CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
6C86 14    50   CALL-TONE-PERIOD
6C87 126   51   TONE PERIOD       = 638
6C89 14    53   CALL-TONE-PERIOD
6C8A 126   54   TONE PERIOD       = 638
6C8C 14    56   CALL-TONE-PERIOD
6C8D 83    57   TONE PERIOD       = 851
6C8F 11    59   CALL-CHANNEL
                ISSUE ON A
6C91 14    61   CALL-TONE-PERIOD
6C92 83    62   TONE PERIOD       = 851
6C94 14    64   CALL-TONE-PERIOD
6C95 204   65   TONE PERIOD       = 716
```

```
6C97 14    67   CALL-TONE-PERIOD
6C98 204   68   TONE PERIOD         = 716
6C9A 14    70   CALL-TONE-PERIOD
6C9B 83    71   TONE PERIOD         = 851
6C9D 14    73   CALL-TONE-PERIOD
6C9E 83    74   TONE PERIOD         = 851
6CA0 11    76   CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
6CA2 14    78   CALL-TONE-PERIOD
6CA3 126   79   TONE PERIOD         = 638
6CA5 14    81   CALL-TONE-PERIOD
6CA6 126   82   TONE PERIOD         = 638
6CA8 14    84   CALL-TONE-PERIOD
6CA9 83    85   TONE PERIOD         = 851
6CAB 14    87   CALL-TONE-PERIOD
6CAC 83    88   TONE PERIOD         = 851
6CAE 11    90   CALL-CHANNEL
                ISSUE ON A
6CB0 14    92   CALL-TONE-PERIOD
6CB1 204   93   TONE PERIOD         = 716
6CB3 11    95   CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
6CB5 14    97   CALL-TONE-PERIOD
6CB6 204   98   TONE PERIOD         = 716
6CB8 11    100  CALL-CHANNEL
                ISSUE ON A
6CBA 14    102  CALL-TONE-PERIOD
6CBB 83    103  TONE PERIOD         = 851
6CBD 11    105  CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
6CBF 14    107  CALL-TONE-PERIOD
6CC0 83    108  TONE PERIOD         = 851
6CC2 14    110  CALL-TONE-PERIOD
6CC3 126   111  TONE PERIOD         = 638
6CC5 14    113  CALL-TONE-PERIOD
6CC6 126   114  TONE PERIOD         = 638
6CC8 14    116  CALL-TONE-PERIOD
6CC9 83    117  TONE PERIOD         = 851
6CCB 11    119  CALL-CHANNEL
                ISSUE ON A
6CCD 14    121  CALL-TONE-PERIOD
6CCE 83    122  TONE PERIOD         = 851
6CD0 14    124  CALL-TONE-PERIOD
6CD1 204   125  TONE PERIOD         = 716
6CD3 14    127  CALL-TONE-PERIOD
6CD4 204   128  TONE PERIOD         = 716
6CD6 14    130  CALL-TONE-PERIOD
6CD7 83    131  TONE PERIOD         = 851
6CD9 14    133  CALL-TONE-PERIOD
6CDA 83    134  TONE PERIOD         = 851
6CDC 11    136  CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
6CDE 14    138  CALL-TONE-PERIOD
6CDF 188   139  TONE PERIOD         = 956
```

```
6CE1 14   141  CALL-TONE-PERIOD
6CE2 188  142  TONE PERIOD         = 956
6CE4 14   144  CALL-TONE-PERIOD
6CE5 126  145  TONE PERIOD         = 638
6CE7 11   147  CALL-CHANNEL
               ISSUE ON A
6CE9 14   149  CALL-TONE-PERIOD
6CEA 126  150  TONE PERIOD         = 638
6CEC 14   152  CALL-TONE-PERIOD
6CED 24   153  TONE PERIOD         = 536
6CEF 14   155  CALL-TONE-PERIOD
6CF0 24   156  TONE PERIOD         = 536
6CF2 14   158  CALL-TONE-PERIOD
6CF3 126  159  TONE PERIOD         = 638
6CF5 14   161  CALL-TONE-PERIOD
6CF6 126  162  TONE PERIOD         = 638
6CF8 11   164  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6CFA 14   166  CALL-TONE-PERIOD
6CFB 188  167  TONE PERIOD         = 956
6CFD 14   169  CALL-TONE-PERIOD
6CFE 188  170  TONE PERIOD         = 956
6D00 14   172  CALL-TONE-PERIOD
6D01 126  173  TONE PERIOD         = 638
6D03 14   175  CALL-TONE-PERIOD
6D04 126  176  TONE PERIOD         = 638
6D06 11   178  CALL-CHANNEL
               ISSUE ON A
6D08 14   180  CALL-TONE-PERIOD
6D09 24   181  TONE PERIOD         = 536
6D0B 11   183  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D0D 14   185  CALL-TONE-PERIOD
6D0E 24   186  TONE PERIOD         = 536
6D10 11   188  CALL-CHANNEL
               ISSUE ON A
6D12 14   190  CALL-TONE-PERIOD
6D13 126  191  TONE PERIOD         = 638
6D15 11   193  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D17 14   195  CALL-TONE-PERIOD
6D18 126  196  TONE PERIOD         = 638
6D1A 14   198  CALL-TONE-PERIOD
6D1B 126  199  TONE PERIOD         = 638
6D1D 14   201  CALL-TONE-PERIOD
6D1E 126  202  TONE PERIOD         = 638
6D20 14   204  CALL-TONE-PERIOD
6D21 83   205  TONE PERIOD         = 851
6D23 11   207  CALL-CHANNEL
               ISSUE ON A
6D25 14   209  CALL-TONE-PERIOD
6D26 83   210  TONE PERIOD         = 851
6D28 14   212  CALL-TONE-PERIOD
6D29 204  213  TONE PERIOD         = 716
6D2B 14   215  CALL-TONE-PERIOD
6D2C 204  216  TONE PERIOD         = 716
6D2E 14   218  CALL-TONE-PERIOD
6D2F 83   219  TONE PERIOD         = 851
6D31 14   221  CALL-TONE-PERIOD
6D32 83   222  TONE PERIOD         = 851
```

```
6D34 11   224  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D36 14   226  CALL-TONE-PERIOD
6D37 83   227  TONE PERIOD         = 851
6D39 14   229  CALL-TONE-PERIOD
6D3A 83   230  TONE PERIOD         = 851
6D3C 14   232  CALL-TONE-PERIOD
6D3D 56   233  TONE PERIOD         = 568
6D3F 11   235  CALL-CHANNEL
               ISSUE ON A
6D41 14   237  CALL-TONE-PERIOD
6D42 56   238  TONE PERIOD         = 568
6D44 14   240  CALL-TONE-PERIOD
6D45 222  241  TONE PERIOD         = 478
6D47 14   243  CALL-TONE-PERIOD
6D48 222  244  TONE PERIOD         = 478
6D4A 14   246  CALL-TONE-PERIOD
6D4B 56   247  TONE PERIOD         = 568
6D4D 14   249  CALL-TONE-PERIOD
6D4E 56   250  TONE PERIOD         = 568
6D50 11   252  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D52 14   254  CALL-TONE-PERIOD
6D53 188  255  TONE PERIOD         = 956
6D55 14   257  CALL-TONE-PERIOD
6D56 188  258  TONE PERIOD         = 956
6D58 14   260  CALL-TONE-PERIOD
6D59 126  261  TONE PERIOD         = 638
6D5B 14   263  CALL-TONE-PERIOD
6D5C 126  264  TONE PERIOD         = 638
6D5E 11   266  CALL-CHANNEL
               ISSUE ON A
6D60 14   268  CALL-TONE-PERIOD
6D61 24   269  TONE PERIOD         = 536
6D63 11   271  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D65 14   273  CALL-TONE-PERIOD
6D66 24   274  TONE PERIOD         = 536
6D68 11   276  CALL-CHANNEL
               ISSUE ON A
6D6A 14   278  CALL-TONE-PERIOD
6D6B 126  279  TONE PERIOD         = 638
6D6D 11   281  CALL-CHANNEL
               ISSUE ON A
               RENDEZVOUS WITH B
               RENDEZVOUS WITH C
6D6F 14   283  CALL-TONE-PERIOD
6D70 126  284  TONE PERIOD         = 638
6D72 14   286  CALL-TONE-PERIOD
6D73 126  287  TONE PERIOD         = 638
6D75 14   289  CALL-TONE-PERIOD
6D76 126  290  TONE PERIOD         = 638
6D78 14   292  CALL-TONE-PERIOD
6D79 83   293  TONE PERIOD         = 851
6D7B 11   295  CALL-CHANNEL
               ISSUE ON A
6D7D 14   297  CALL-TONE-PERIOD
6D7E 83   298  TONE PERIOD         = 851
6D80 14   300  CALL-TONE-PERIOD
6D81 204  301  TONE PERIOD         = 716
```

```
6D83 14    303  CALL-TONE-PERIOD
6D84 204   304  TONE PERIOD          = 716
6D86 14    306  CALL-TONE-PERIOD
6D87 83    307  TONE PERIOD          = 851
6D89 14    309  CALL-TONE-PERIOD
6D8A 83    310  TONE PERIOD          = 851
6D8C 14    312  CALL-TONE-PERIOD
6D8D 126   313  TONE PERIOD          = 638
6D8F 14    315  CALL-TONE-PERIOD
6D90 126   316  TONE PERIOD          = 638
6D92 14    318  CALL-TONE-PERIOD
6D93 83    319  TONE PERIOD          = 851
6D95 14    321  CALL-TONE-PERIOD
6D96 83    322  TONE PERIOD          = 851
6D98 14    324  CALL-TONE-PERIOD
6D99 204   325  TONE PERIOD          = 716
6D9B 14    327  CALL-TONE-PERIOD
6D9C 204   328  TONE PERIOD          = 716
6D9E 14    330  CALL-TONE-PERIOD
6D9F 83    331  TONE PERIOD          = 851
6DA1 14    333  CALL-TONE-PERIOD
6DA2 83    334  TONE PERIOD          = 851
6DA4 9     336  JUMP
6DA5 48    337  PC ADDRESS           = 48


CHANNEL B


6E44 1     1    WAIT-SOUND
                ISSUE ON B
                RENDEZVOUS WITH A
                RENDEZVOUS WITH C
6E46 2     3    AMPLITUDE ENVELOPE= 2
6E47 0     4    TONE ENVELOPE        = 0
6E48 213   5    TONE PERIOD          = 213
6E4A 0     7    NOISE PERIOD         = 0
6E4B 0     8    INITIAL VOLUME       = 0
6E4C 23    9    DURATION             = 23
6E4E 11    11   CALL-CHANNEL
                ISSUE ON B
                RENDEZVOUS WITH A
                RENDEZVOUS WITH C
6E50 14    13   CALL-TONE-PERIOD
6E51 28    14   TONE PERIOD          = 284
6E53 14    16   CALL-TONE-PERIOD
6E54 28    17   TONE PERIOD          = 284
6E56 14    19   CALL-TONE-PERIOD
6E57 253   20   TONE PERIOD          = 253
6E59 14    22   CALL-TONE-PERIOD
6E5A 28    23   TONE PERIOD          = 284
6E5C 14    25   CALL-TONE-PERIOD
6E5D 28    26   TONE PERIOD          = 284
6E5F 14    28   CALL-TONE-PERIOD
6E60 213   29   TONE PERIOD          = 213
6E62 14    31   CALL-TONE-PERIOD
6E63 28    32   TONE PERIOD          = 284
6E65 14    34   CALL-TONE-PERIOD
6E66 213   35   TONE PERIOD          = 213
6E68 14    37   CALL-TONE-PERIOD
6E69 213   38   TONE PERIOD          = 213
6E6B 14    40   CALL-TONE-PERIOD
6E6C 28    41   TONE PERIOD          = 284
6E6E 14    43   CALL-TONE-PERIOD
6E6F 28    44   TONE PERIOD          = 284
6E71 14    46   CALL-TONE-PERIOD
```

```
6E72 253   47   TONE PERIOD          = 253
6E74 14    49   CALL-TONE-PERIOD
6E75 213   50   TONE PERIOD          = 213
6E77 14    52   CALL-TONE-PERIOD
6E78 213   53   TONE PERIOD          = 213
6E7A 14    55   CALL-TONE-PERIOD
6E7B 190   56   TONE PERIOD          = 190
6E7D 14    58   CALL-TONE-PERIOD
6E7E 213   59   TONE PERIOD          = 213
6E80 14    61   CALL-TONE-PERIOD
6E81 213   62   TONE PERIOD          = 213
6E83 14    64   CALL-TONE-PERIOD
6E84 159   65   TONE PERIOD          = 159
6E86 14    67   CALL-TONE-PERIOD
6E87 213   68   TONE PERIOD          = 213
6E89 14    70   CALL-TONE-PERIOD
6E8A 159   71   TONE PERIOD          = 159
6E8C 14    73   CALL-TONE-PERIOD
6E8D 213   74   TONE PERIOD          = 213
6E8F 14    76   CALL-TONE-PERIOD
6E90 28    77   TONE PERIOD          = 284
6E92 14    79   CALL-TONE-PERIOD
6E93 28    80   TONE PERIOD          = 284
6E95 14    82   CALL-TONE-PERIOD
6E96 253   83   TONE PERIOD          = 253
6E98 14    85   CALL-TONE-PERIOD
6E99 190   86   TONE PERIOD          = 190
6E9B 14    88   CALL-TONE-PERIOD
6E9C 190   89   TONE PERIOD          = 190
6E9E 14    91   CALL-TONE-PERIOD
6E9F 169   92   TONE PERIOD          = 169
6EA1 14    94   CALL-TONE-PERIOD
6EA2 213   95   TONE PERIOD          = 213
6EA4 14    97   CALL-TONE-PERIOD
6EA5 213   98   TONE PERIOD          = 213
6EA7 14    100  CALL-TONE-PERIOD
6EA8 159   101  TONE PERIOD          = 159
6EAA 14    103  CALL-TONE-PERIOD
6EAB 213   104  TONE PERIOD          = 213
6EAD 14    106  CALL-TONE-PERIOD
6EAE 159   107  TONE PERIOD          = 159
6EB0 14    109  CALL-TONE-PERIOD
6EB1 213   110  TONE PERIOD          = 213
6EB3 14    112  CALL-TONE-PERIOD
6EB4 28    113  TONE PERIOD          = 284
6EB6 14    115  CALL-TONE-PERIOD
6EB7 28    116  TONE PERIOD          = 284
6EB9 14    118  CALL-TONE-PERIOD
6EBA 253   119  TONE PERIOD          = 253
6EBC 9     121  JUMP
6EBD 11    122  PC ADDRESS           = 11


CHANNEL C


6DAC 1     1    WAIT-SOUND
                ISSUE ON C
                RENDEZVOUS WITH A
                RENDEZVOUS WITH B
6DAE 2     3    AMPLITUDE ENVELOPE= 2
6DAF 0     4    TONE ENVELOPE        = 0
6DB0 134   5    TONE PERIOD          = 134
6DB2 0     7    NOISE PERIOD         = 0
6DB3 0     8    INITIAL VOLUME       = 0
6DB4 23    9    DURATION             = 23
```

```
6DB6 11     11    CALL-CHANNEL
                  ISSUE ON C
                  RENDEZVOUS WITH A
                  RENDEZVOUS WITH B
6DB8 14     13    CALL-TONE-PERIOD
6DB9 179    14    TONE PERIOD          = 179
6DBB 14     16    CALL-TONE-PERIOD
6DBC 179    17    TONE PERIOD          = 179
6DBE 15     19    CALL-NOISE-PERIOD
6DBF 1      20    NOISE PERIOD         = 1
6DC0 14     21    CALL-TONE-PERIOD
6DC1 159    22    TONE PERIOD          = 159
6DC3 15     24    CALL-NOISE-PERIOD
6DC4 0      25    NOISE PERIOD         = 0
6DC5 14     26    CALL-TONE-PERIOD
6DC6 179    27    TONE PERIOD          = 179
6DC8 14     29    CALL-TONE-PERIOD
6DC9 179    30    TONE PERIOD          = 179
6DCB 14     32    CALL-TONE-PERIOD
6DCC 213    33    TONE PERIOD          = 213
6DCE 14     35    CALL-TONE-PERIOD
6DCF 179    36    TONE PERIOD          = 179
6DD1 14     38    CALL-TONE-PERIOD
6DD2 213    39    TONE PERIOD          = 213
6DD4 14     41    CALL-TONE-PERIOD
6DD5 213    42    TONE PERIOD          = 213
6DD7 14     44    CALL-TONE-PERIOD
6DD8 179    45    TONE PERIOD          = 179
6DDA 14     47    CALL-TONE-PERIOD
6DDB 179    48    TONE PERIOD          = 179
6DDD 15     50    CALL-NOISE-PERIOD
6DDE 1      51    NOISE PERIOD         = 1
6DDF 14     52    CALL-TONE-PERIOD
6DE0 159    53    TONE PERIOD          = 159
6DE2 15     55    CALL-NOISE-PERIOD
6DE3 0      56    NOISE PERIOD         = 0
6DE4 14     57    CALL-TONE-PERIOD
6DE5 134    58    TONE PERIOD          = 134
6DE7 14     60    CALL-TONE-PERIOD
6DE8 134    61    TONE PERIOD          = 134
6DEA 15     63    CALL-NOISE-PERIOD
6DEB 1      64    NOISE PERIOD         = 1
6DEC 14     65    CALL-TONE-PERIOD
6DED 119    66    TONE PERIOD          = 119
6DEF 15     68    CALL-NOISE-PERIOD
6DF0 0      69    NOISE PERIOD         = 0
6DF1 14     70    CALL-TONE-PERIOD
6DF2 134    71    TONE PERIOD          = 134
6DF4 14     73    CALL-TONE-PERIOD
6DF5 134    74    TONE PERIOD          = 134
6DF7 14     76    CALL-TONE-PERIOD
6DF8 159    77    TONE PERIOD          = 159
6DFA 14     79    CALL-TONE-PERIOD
6DFB 134    80    TONE PERIOD          = 134
6DFD 14     82    CALL-TONE-PERIOD
6DFE 159    83    TONE PERIOD          = 159
6E00 14     85    CALL-TONE-PERIOD
6E01 213    86    TONE PERIOD          = 213
6E03 14     88    CALL-TONE-PERIOD
6E04 179    89    TONE PERIOD          = 179
6E06 14     91    CALL-TONE-PERIOD
6E07 179    92    TONE PERIOD          = 179
6E09 15     94    CALL-NOISE-PERIOD
6E0A 1      95    NOISE PERIOD         = 1
6E0B 14     96    CALL-TONE-PERIOD
6E0C 159    97    TONE PERIOD          = 159
```

```
6E0E 15     99    CALL-NOISE-PERIOD
6E0F 0      100   NOISE PERIOD         = 0
6E10 14     101   CALL-TONE-PERIOD
6E11 119    102   TONE PERIOD          = 119
6E13 14     104   CALL-TONE-PERIOD
6E14 119    105   TONE PERIOD          = 119
6E16 15     107   CALL-NOISE-PERIOD
6E17 1      108   NOISE PERIOD         = 1
6E18 14     109   CALL-TONE-PERIOD
6E19 213    110   TONE PERIOD          = 213
6E1B 15     112   CALL-NOISE-PERIOD
6E1C 0      113   NOISE PERIOD         = 0
6E1D 14     114   CALL-TONE-PERIOD
6E1E 134    115   TONE PERIOD          = 134
6E20 14     117   CALL-TONE-PERIOD
6E21 134    118   TONE PERIOD          = 134
6E23 14     120   CALL-TONE-PERIOD
6E24 159    121   TONE PERIOD          = 159
6E26 14     123   CALL-TONE-PERIOD
6E27 134    124   TONE PERIOD          = 134
6E29 14     126   CALL-TONE-PERIOD
6E2A 159    127   TONE PERIOD          = 159
6E2C 14     129   CALL-TONE-PERIOD
6E2D 213    130   TONE PERIOD          = 213
6E2F 14     132   CALL-TONE-PERIOD
6E30 179    133   TONE PERIOD          = 179
6E32 14     135   CALL-TONE-PERIOD
6E33 179    136   TONE PERIOD          = 179
6E35 15     138   CALL-NOISE-PERIOD
6E36 1      139   NOISE PERIOD         = 1
6E37 14     140   CALL-TONE-PERIOD
6E38 159    141   TONE PERIOD          = 159
6E3A 15     143   CALL-NOISE-PERIOD
6E3B 0      144   NOISE PERIOD         = 0
6E3C 9      145   JUMP
6E3D 11     146   PC ADDRESS           = 11
```

THE 'HUNCHBACK' SCREEN

```
CHANNEL  A


68DF 1     1    WAIT-SOUND
                ISSUE ON A
68E1 1     3    AMPLITUDE ENVELOPE= 1
68E2 1     4    TONE ENVELOPE     = 1
68E3 90    5    TONE PERIOD       = 602
68E5 0     7    NOISE PERIOD      = 0
68E6 0     8    INITIAL VOLUME    = 0
68E7 20    9    DURATION          = 20
68E9 2     11   RESET
68EA 6     12   AMP-ENV
68EB 1     13   AMPLITUDE ENVELOPE= 1
68EC 5     14   NO OF SECTIONS    = 5
68ED 2     15   STEP COUNT        = 2
68EE 5     16   STEP SIZE         = 5
68EF 1     17   PAUSE TIME        = 1
68F0 1     18   STEP COUNT        = 1
68F1 5     19   STEP SIZE         = 5
68F2 1     20   PAUSE TIME        = 1
68F3 1     21   STEP COUNT        = 1
68F4 0     22   STEP SIZE         = 0
68F5 5     23   PAUSE TIME        = 5
68F6 10    24   STEP COUNT        = 10
68F7 255   25   STEP SIZE         = 255
68F8 1     26   PAUSE TIME        = 1
68F9 5     27   STEP COUNT        = 5
68FA 255   28   STEP SIZE         = 255
68FB 10    29   PAUSE TIME        = 10
68FC 6     30   AMP-ENV
68FD 2     31   AMPLITUDE ENVELOPE= 2
68FE 5     32   NO OF SECTIONS    = 5
68FF 2     33   STEP COUNT        = 2
6900 5     34   STEP SIZE         = 5
6901 1     35   PAUSE TIME        = 1
6902 1     36   STEP COUNT        = 1
6903 5     37   STEP SIZE         = 5
6904 1     38   PAUSE TIME        = 1
6905 1     39   STEP COUNT        = 1
6906 0     40   STEP SIZE         = 0
6907 5     41   PAUSE TIME        = 5
6908 4     42   STEP COUNT        = 4
6909 255   43   STEP SIZE         = 255
690A 1     44   PAUSE TIME        = 1
690B 3     45   STEP COUNT        = 3
690C 254   46   STEP SIZE         = 254
690D 5     47   PAUSE TIME        = 5
690E 6     48   AMP-ENV
690F 4     49   AMPLITUDE ENVELOPE= 4
6910 1     50   NO OF SECTIONS    = 1
6911 1     51   STEP COUNT        = 1
6912 0     52   STEP SIZE         = 0
6913 100   53   PAUSE TIME        = 100
6914 7     54   TONE-ENV
6915 255   55   TONE ENVELOPE     = 255
6916 3     56   NO OF SECTIONS    = 3
6917 2     57   STEP COUNT        = 2
6918 2     58   STEP SIZE         = 2
6919 2     59   PAUSE TIME        = 2
691A 4     60   STEP COUNT        = 4
691B 254   61   STEP SIZE         = 254
691C 2     62   PAUSE TIME        = 2
691D 2     63   STEP COUNT        = 2
691E 2     64   STEP SIZE         = 2
691F 2     65   PAUSE TIME        = 2
6920 6     66   AMP-ENV
6921 3     67   AMPLITUDE ENVELOPE= 3
6922 2     68   NO OF SECTIONS    = 2
                HARDWARE ENVELOPE
6923 138   69   ENVELOPE SHAPE    = 138
6924 142   70   ENVELOPE PERIOD   = 1422
6926 1     72   STEP COUNT        = 1
6927 0     73   STEP SIZE         = 0
6928 20    74   PAUSE TIME        = 20
6929 11    75   CALL-CHANNEL
                ISSUE ON A
                RENDEZVOUS WITH B
                RENDEZVOUS WITH C
692B 18    77   CALL-TONE-DURATION
692C 100   78   DURATION          = 100
692E 56    80   TONE PERIOD       = 568
6930 11    82   CALL-CHANNEL
                ISSUE ON A
6932 18    84   CALL-TONE-DURATION
6933 20    85   DURATION          = 20
6935 56    87   TONE PERIOD       = 568
6937 18    89   CALL-TONE-DURATION
6938 100   90   DURATION          = 100
693A 188   92   TONE PERIOD       = 956
693C 18    94   CALL-TONE-DURATION
693D 20    95   DURATION          = 20
693F 188   97   TONE PERIOD       = 956
6941 18    99   CALL-TONE-DURATION
6942 100   100  DURATION          = 100
6944 83    102  TONE PERIOD       = 851
6946 18    104  CALL-TONE-DURATION
6947 20    105  DURATION          = 20
6949 83    107  TONE PERIOD       = 851
694B 18    109  CALL-TONE-DURATION
694C 60    110  DURATION          = 60
694E 123   112  TONE PERIOD       = 379
6950 18    114  CALL-TONE-DURATION
6951 20    115  DURATION          = 20
6953 246   117  TONE PERIOD       = 758
6955 14    119  CALL-TONE-PERIOD
6956 164   120  TONE PERIOD       = 676
6958 14    122  CALL-TONE-PERIOD
6959 90    123  TONE PERIOD       = 602
695B 18    125  CALL-TONE-DURATION
695C 100   126  DURATION          = 100
695E 56    128  TONE PERIOD       = 568
6960 18    130  CALL-TONE-DURATION
6961 20    131  DURATION          = 20
6963 56    133  TONE PERIOD       = 568
6965 18    135  CALL-TONE-DURATION
6966 100   136  DURATION          = 100
6968 188   138  TONE PERIOD       = 956
696A 18    140  CALL-TONE-DURATION
696B 20    141  DURATION          = 20
696D 188   143  TONE PERIOD       = 956
696F 18    145  CALL-TONE-DURATION
6970 100   146  DURATION          = 100
6972 83    148  TONE PERIOD       = 851
6974 18    150  CALL-TONE-DURATION
6975 20    151  DURATION          = 20
6977 83    153  TONE PERIOD       = 851
```

```
6979 14   155  CALL-TONE-PERIOD
697A 123  156  TONE PERIOD          = 379
697C 14   158  CALL-TONE-PERIOD
697D 246  159  TONE PERIOD          = 758
697F 14   161  CALL-TONE-PERIOD
6980 204  162  TONE PERIOD          = 716
6982 14   164  CALL-TONE-PERIOD
6983 164  165  TONE PERIOD          = 676
6985 14   167  CALL-TONE-PERIOD
6986 126  168  TONE PERIOD          = 638
6988 14   170  CALL-TONE-PERIOD
6989 90   171  TONE PERIOD          = 602
698B 18   173  CALL-TONE-DURATION
698C 100  174  DURATION             = 100
698E 56   176  TONE PERIOD          = 568
6990 18   178  CALL-TONE-DURATION
6991 20   179  DURATION             = 20
6993 56   181  TONE PERIOD          = 568
6995 18   183  CALL-TONE-DURATION
6996 100  184  DURATION             = 100
6998 188  186  TONE PERIOD          = 956
699A 18   188  CALL-TONE-DURATION
699B 20   189  DURATION             = 20
699D 188  191  TONE PERIOD          = 956
699F 18   193  CALL-TONE-DURATION
69A0 100  194  DURATION             = 100
69A2 83   196  TONE PERIOD          = 851
69A4 18   198  CALL-TONE-DURATION
69A5 20   199  DURATION             = 20
69A7 83   201  TONE PERIOD          = 851
69A9 18   203  CALL-TONE-DURATION
69AA 100  204  DURATION             = 100
69AC 123  206  TONE PERIOD          = 379
69AE 18   208  CALL-TONE-DURATION
69AF 20   209  DURATION             = 20
69B1 123  211  TONE PERIOD          = 379
69B3 18   213  CALL-TONE-DURATION
69B4 60   214  DURATION             = 60
69B6 56   216  TONE PERIOD          = 568
69B8 14   218  CALL-TONE-PERIOD
69B9 126  219  TONE PERIOD          = 638
69BB 18   221  CALL-TONE-DURATION
69BC 40   222  DURATION             = 40
69BE 123  224  TONE PERIOD          = 379
69C0 18   226  CALL-TONE-DURATION
69C1 20   227  DURATION             = 20
69C3 83   229  TONE PERIOD          = 851
69C5 18   231  CALL-TONE-DURATION
69C6 40   232  DURATION             = 40
69C8 188  234  TONE PERIOD          = 956
69CA 18   236  CALL-TONE-DURATION
69CB 20   237  DURATION             = 20
69CD 56   239  TONE PERIOD          = 568
69CF 18   241  CALL-TONE-DURATION
69D0 40   242  DURATION             = 40
69D2 83   244  TONE PERIOD          = 851
69D4 18   246  CALL-TONE-DURATION
69D5 20   247  DURATION             = 20
69D7 123  249  TONE PERIOD          = 379
69D9 18   251  CALL-TONE-DURATION
69DA 40   252  DURATION             = 40
69DC 188  254  TONE PERIOD          = 956
69DE 18   256  CALL-TONE-DURATION
69DF 60   257  DURATION             = 60
69E1 56   259  TONE PERIOD          = 568
69E3 18   261  CALL-TONE-DURATION
69E4 80   262  DURATION             = 80
69E6 123  264  TONE PERIOD          = 379
69E8 9    266  JUMP
69E9 75   267  PC ADDRESS           = 75
```

```
CHANNEL B

67F2 1     1    WAIT-SOUND
                ISSUE ON B
67F4 2     3    AMPLITUDE ENVELOPE= 2
67F5 1     4    TONE ENVELOPE      = 1
67F6 142   5    TONE PERIOD        = 142
67F8 0     7    NOISE PERIOD       = 0
67F9 0     8    INITIAL VOLUME     = 0
67FA 60    9    DURATION           = 60
67FC 11    11   CALL-CHANNEL
                ISSUE ON B
                RENDEZVOUS WITH A
                RENDEZVOUS WITH C
67FE 12    13   CALL-AMP-ENV
67FF 4     14   AMPLITUDE ENVELOPE= 4
6800 18    15   CALL-TONE-DURATION
6801 40    16   DURATION           = 40
6803 119   18   TONE PERIOD        = 119
6805 11    20   CALL-CHANNEL
                ISSUE ON B
6807 12    22   CALL-AMP-ENV
6808 2     23   AMPLITUDE ENVELOPE= 2
6809 18    24   CALL-TONE-DURATION
680A 20    25   DURATION           = 20
680C 119   27   TONE PERIOD        = 119
680E 18    29   CALL-TONE-DURATION
680F 60    30   DURATION           = 60
6811 142   32   TONE PERIOD        = 142
6813 18    34   CALL-TONE-DURATION
6814 40    35   DURATION           = 40
6816 159   37   TONE PERIOD        = 159
6818 18    39   CALL-TONE-DURATION
6819 20    40   DURATION           = 20
681B 190   42   TONE PERIOD        = 190
681D 18    44   CALL-TONE-DURATION
681E 40    45   DURATION           = 40
6820 213   47   TONE PERIOD        = 213
6822 18    49   CALL-TONE-DURATION
6823 20    50   DURATION           = 20
6825 239   52   TONE PERIOD        = 239
6827 18    54   CALL-TONE-DURATION
6828 60    55   DURATION           = 60
682A 213   57   TONE PERIOD        = 213
682C 18    59   CALL-TONE-DURATION
682D 40    60   DURATION           = 40
682F 213   62   TONE PERIOD        = 213
6831 18    64   CALL-TONE-DURATION
6832 20    65   DURATION           = 20
6834 239   67   TONE PERIOD        = 239
6836 18    69   CALL-TONE-DURATION
6837 40    70   DURATION           = 40
6839 213   72   TONE PERIOD        = 213
683B 18    74   CALL-TONE-DURATION
683C 120   75   DURATION           = 120
683E 190   77   TONE PERIOD        = 190
6840 18    79   CALL-TONE-DURATION
6841 20    80   DURATION           = 20
6843 119   82   TONE PERIOD        = 119
```

```
6845 18    84    CALL-TONE-DURATION
6846 60    85    DURATION           = 60
6848 142   87    TONE PERIOD        = 142
684A 18    89    CALL-TONE-DURATION
684B 40    90    DURATION           = 40
684D 159   92    TONE PERIOD        = 159
684F 18    94    CALL-TONE-DURATION
6850 20    95    DURATION           = 20
6852 190   97    TONE PERIOD        = 190
6854 18    99    CALL-TONE-DURATION
6855 40    100   DURATION           = 40
6857 213   102   TONE PERIOD        = 213
6859 18    104   CALL-TONE-DURATION
685A 20    105   DURATION           = 20
685C 239   107   TONE PERIOD        = 239
685E 18    109   CALL-TONE-DURATION
685F 40    110   DURATION           = 40
6861 213   112   TONE PERIOD        = 213
6863 18    114   CALL-TONE-DURATION
6864 20    115   DURATION           = 20
6866 239   117   TONE PERIOD        = 239
6868 18    119   CALL-TONE-DURATION
6869 40    120   DURATION           = 40
686B 213   122   TONE PERIOD        = 213
686D 18    124   CALL-TONE-DURATION
686E 180   125   DURATION           = 180
6870 190   127   TONE PERIOD        = 190
6872 18    129   CALL-TONE-DURATION
6873 20    130   DURATION           = 20
6875 80    132   TONE PERIOD        = 80
6877 18    134   CALL-TONE-DURATION
6878 60    135   DURATION           = 60
687A 95    137   TONE PERIOD        = 95
687C 18    139   CALL-TONE-DURATION
687D 40    140   DURATION           = 40
687F 106   142   TONE PERIOD        = 106
6881 18    144   CALL-TONE-DURATION
6882 20    145   DURATION           = 20
6884 127   147   TONE PERIOD        = 127
6886 18    149   CALL-TONE-DURATION
6887 40    150   DURATION           = 40
6889 142   152   TONE PERIOD        = 142
688B 18    154   CALL-TONE-DURATION
688C 20    155   DURATION           = 20
688E 159   157   TONE PERIOD        = 159
6890 18    159   CALL-TONE-DURATION
6891 60    160   DURATION           = 60
6893 142   162   TONE PERIOD        = 142
6895 14    164   CALL-TONE-PERIOD
6896 142   165   TONE PERIOD        = 142
6898 18    167   CALL-TONE-DURATION
6899 40    168   DURATION           = 40
689B 142   170   TONE PERIOD        = 142
689D 18    172   CALL-TONE-DURATION
689E 80    173   DURATION           = 80
68A0 127   175   TONE PERIOD        = 127
68A2 18    177   CALL-TONE-DURATION
68A3 60    178   DURATION           = 60
68A5 142   180   TONE PERIOD        = 142
68A7 14    182   CALL-TONE-PERIOD
68A8 159   183   TONE PERIOD        = 159
68AA 18    185   CALL-TONE-DURATION
68AB 40    186   DURATION           = 40
68AD 190   188   TONE PERIOD        = 190
68AF 18    190   CALL-TONE-DURATION
68B0 20    191   DURATION           = 20
```

```
68B2 213   193   TONE PERIOD        = 213
68B4 18    195   CALL-TONE-DURATION
68B5 40    196   DURATION           = 40
68B7 239   198   TONE PERIOD        = 239
68B9 18    200   CALL-TONE-DURATION
68BA 20    201   DURATION           = 20
68BC 28    203   TONE PERIOD        = 284
68BE 18    205   CALL-TONE-DURATION
68BF 40    206   DURATION           = 40
68C1 213   208   TONE PERIOD        = 213
68C3 18    210   CALL-TONE-DURATION
68C4 20    211   DURATION           = 20
68C6 190   213   TONE PERIOD        = 190
68C8 18    215   CALL-TONE-DURATION
68C9 40    216   DURATION           = 40
68CB 237   218   TONE PERIOD        = 237
68CD 18    220   CALL-TONE-DURATION
68CE 60    221   DURATION           = 60
68D0 28    223   TONE PERIOD        = 284
68D2 18    225   CALL-TONE-DURATION
68D3 80    226   DURATION           = 80
68D5 150   228   TONE PERIOD        = 150
68D7 9     230   JUMP
68D8 11    231   PC ADDRESS         = 11


CHANNEL C


6F0F 1     1     WAIT-SOUND
                 ISSUE ON C
6F11 3     3     AMPLITUDE ENVELOPE= 3
6F12 3     4     TONE ENVELOPE      = 3
6F13 71    5     TONE PERIOD        = 71
6F15 0     7     NOISE PERIOD       = 0
6F16 0     8     INITIAL VOLUME     = 0
6F17 60    9     DURATION           = 60
6F19 12    11    CALL-AMP-ENV
6F1A 4     12    AMPLITUDE ENVELOPE= 4
6F1B 11    13    CALL-CHANNEL
                 ISSUE ON C
                 RENDEZVOUS WITH A
                 RENDEZVOUS WITH B
6F1D 18    15    CALL-TONE-DURATION
6F1E 40    16    DURATION           = 40
6F20 119   18    TONE PERIOD        = 119
6F22 12    20    CALL-AMP-ENV
6F23 3     21    AMPLITUDE ENVELOPE= 3
6F24 11    22    CALL-CHANNEL
                 ISSUE ON C
6F26 18    24    CALL-TONE-DURATION
6F27 20    25    DURATION           = 20
6F29 60    27    TONE PERIOD        = 60
6F2B 18    29    CALL-TONE-DURATION
6F2C 60    30    DURATION           = 60
6F2E 71    32    TONE PERIOD        = 71
6F30 18    34    CALL-TONE-DURATION
6F31 40    35    DURATION           = 40
6F33 80    37    TONE PERIOD        = 80
6F35 18    39    CALL-TONE-DURATION
6F36 20    40    DURATION           = 20
6F38 95    42    TONE PERIOD        = 95
6F3A 18    44    CALL-TONE-DURATION
6F3B 40    45    DURATION           = 40
6F3D 106   47    TONE PERIOD        = 106
6F3F 18    49    CALL-TONE-DURATION
```

```
6F40 20    50   DURATION              = 20
6F42 119   52   TONE PERIOD           = 119
6F44 18    54   CALL-TONE-DURATION
6F45 60    55   DURATION              = 60
6F47 106   57   TONE PERIOD           = 106
6F49 18    59   CALL-TONE-DURATION
6F4A 40    60   DURATION              = 40
6F4C 106   62   TONE PERIOD           = 106
6F4E 18    64   CALL-TONE-DURATION
6F4F 20    65   DURATION              = 20
6F51 119   67   TONE PERIOD           = 119
6F53 18    69   CALL-TONE-DURATION
6F54 40    70   DURATION              = 40
6F56 106   72   TONE PERIOD           = 106
6F58 19    74   CALL-TONE-DURATION
6F59 120   75   DURATION              = 120
6F5B 95    77   TONE PERIOD           = 95
6F5D 18    79   CALL-TONE-DURATION
6F5E 20    80   DURATION              = 20
6F60 60    82   TONE PERIOD           = 60
6F62 18    84   CALL-TONE-DURATION
6F63 60    85   DURATION              = 60
6F65 71    87   TONE PERIOD           = 71
6F67 18    89   CALL-TONE-DURATION
6F68 40    90   DURATION              = 40
6F6A 80    92   TONE PERIOD           = 80
6F6C 18    94   CALL-TONE-DURATION
6F6D 20    95   DURATION              = 20
6F6F 95    97   TONE PERIOD           = 95
6F71 18    99   CALL-TONE-DURATION
6F72 40    100  DURATION              = 40
6F74 106   102  TONE PERIOD           = 106
6F76 18    104  CALL-TONE-DURATION
6F77 20    105  DURATION              = 20
6F79 119   107  TONE PERIOD           = 119
6F7B 18    109  CALL-TONE-DURATION
6F7C 40    110  DURATION              = 40
6F7E 106   112  TONE PERIOD           = 106
6F80 18    114  CALL-TONE-DURATION
6F81 20    115  DURATION              = 20
6F83 119   117  TONE PERIOD           = 119
6F85 18    119  CALL-TONE-DURATION
6F86 40    120  DURATION              = 40
6F88 106   122  TONE PERIOD           = 106
6F8A 18    124  CALL-TONE-DURATION
6F8B 180   125  DURATION              = 180
6F8D 95    127  TONE PERIOD           = 95
6F8F 18    129  CALL-TONE-DURATION
6F90 20    130  DURATION              = 20
6F92 60    132  TONE PERIOD           = 60
6F94 18    134  CALL-TONE-DURATION
6F95 60    135  DURATION              = 60
6F97 71    137  TONE PERIOD           = 71
6F99 18    139  CALL-TONE-DURATION
6F9A 40    140  DURATION              = 40
6F9C 80    142  TONE PERIOD           = 80
6F9E 18    144  CALL-TONE-DURATION
6F9F 20    145  DURATION              = 20
6FA1 95    147  TONE PERIOD           = 95
6FA3 18    149  CALL-TONE-DURATION
6FA4 40    150  DURATION              = 40
6FA6 106   152  TONE PERIOD           = 106
6FA8 18    154  CALL-TONE-DURATION
6FA9 20    155  DURATION              = 20
6FAB 119   157  TONE PERIOD           = 119
6FAD 18    159  CALL-TONE-DURATION
```

```
6FAE 60    160  DURATION              = 60
6FB0 106   162  TONE PERIOD           = 106
6FB2 18    164  CALL-TONE-DURATION
6FB3 40    165  DURATION              = 40
6FB5 106   167  TONE PERIOD           = 106
6FB7 18    169  CALL-TONE-DURATION
6FB8 20    170  DURATION              = 20
6FBA 119   172  TONE PERIOD           = 119
6FBC 18    174  CALL-TONE-DURATION
6FBD 40    175  DURATION              = 40
6FBF 106   177  TONE PERIOD           = 106
6FC1 18    179  CALL-TONE-DURATION
6FC2 80    180  DURATION              = 80
6FC4 95    182  TONE PERIOD           = 95
6FC6 18    184  CALL-TONE-DURATION
6FC7 60    185  DURATION              = 60
6FC9 71    187  TONE PERIOD           = 71
6FCB 14    189  CALL-TONE-PERIOD
6FCC 80    190  TONE PERIOD           = 80
6FCE 18    192  CALL-TONE-DURATION
6FCF 40    193  DURATION              = 40
6FD1 95    195  TONE PERIOD           = 95
6FD3 18    197  CALL-TONE-DURATION
6FD4 20    198  DURATION              = 20
6FD6 106   200  TONE PERIOD           = 106
6FD8 18    202  CALL-TONE-DURATION
6FD9 40    203  DURATION              = 40
6FDB 119   205  TONE PERIOD           = 119
6FDD 18    207  CALL-TONE-DURATION
6FDE 20    208  DURATION              = 20
6FE0 142   210  TONE PERIOD           = 142
6FE2 18    212  CALL-TONE-DURATION
6FE3 40    213  DURATION              = 40
6FE5 106   215  TONE PERIOD           = 106
6FE7 18    217  CALL-TONE-DURATION
6FE8 20    218  DURATION              = 20
6FEA 95    220  TONE PERIOD           = 95
6FEC 18    222  CALL-TONE-DURATION
6FED 40    223  DURATION              = 40
6FEF 119   225  TONE PERIOD           = 119
6FF1 18    227  CALL-TONE-DURATION
6FF2 60    228  DURATION              = 60
6FF4 142   230  TONE PERIOD           = 142
6FF6 18    232  CALL-TONE-DURATION
6FF7 80    233  DURATION              = 80
6FF9 119   235  TONE PERIOD           = 119
6FFB 9     237  JUMP
6FFC 11    238  PC ADDRESS            = 11
```

TRACKING SPRITES

Sound handling and tracking sprites are the two most difficult to use facilities of Laser BASIC and we do not recommend tackling them until you have gained a fairly good familiarisation with the package. Tracking sprites in particular have only modest crash protection and it is usually very difficult for us to provide answers to technical queries arising from their use or misuse. Having said all that, if you do spend the time and effort then the results can be very satisfying indeed. In this final part of "GETTING STARTED" we'll look at a few examples of the use of tracking sprites.

A tracking sprite , in this context, is a sprite which contains a program that is used to control the movement of a graphics sprite or sprites. The internal format is described in the section "The Laser BASIC commands in detail". Since tracking sprites are normally used to move sprites around predetermined tracks it is normally required to start by physically putting a sprite onto the screen in the same way as PTXR for example, is normally used before XMOV or XBNC. This is normally carried out with the TPUT command which as well as physically placing the image onto the screen also initialises some of the trackers internal parameters. The tracker maintains its own record of where it last placed a sprite and which sprite it last placed as well as what type of 'PUT' it is using and whereabouts within itself it is executing. All of these parameters can be set up with POKE's but it is probably easier to set them up with a TPUT.

In this first example we'll set up a very simple tracking sprite that just bounces a sprite around a predefined window and loops within itself. Type "NEW" to clear the current program and then enter and RUN the following:

```
5 ' EXAMPLE OF TRACKING SPRITES
10 X%=0:Y%=0:|SPN,40:|ISPR,@X%,@X%,@X%,@Y%
20 DATA 2,0,4,4,181,162,0,0
30 FOR I%=Y%+5 TO Y%+12 :READ X%:POKE I%,X%:NEXT I%
40 |SET,4:|KEY,51:|SPN,51 :|COL,0  :|ROW,75:|HGT,50:|LEN,80:|BWST
50 |HGT,4:|LEN,2 :|COL,40 :|ROW,100:|SP1,46:|SP2,47:|SP3,48:|SP4,49
60 |SET,0:|COL,40:|ROW,100:|KEY,40 :|SP1,46:|SCLS  :|TPUT,1
70 |TMOV,1000,1
```

Sprite 46 should appear at column 15, row 25.

Line 10 Initialises X% and Y% and then interrogates sprite 40 to find its starting address which is assigned to Y%. Sprite 40 is 13 bytes in length.

Line 20 Data for the sprite tracking program.

Line 30 POKEs in the tracking sprite program. The first five bytes of sprite 40 are utilised by the tracker program itself and the data has the following significance:

1st byte = 2 This is the move type - XOR

2nd byte = 0 This is the first byte of the program and tells the tracker that the next byte is to be a control code.

3rd byte = 4 This code tells the tracker that the next byte will be a SET number and the next two bytes after that, the address of a Laser BASIC command.

4th byte = 4 This is the SET, which is SET 4.

5th byte = 181 These two bytes are the execution address of
6th byte = 162 the Laser BASIC command XBNC and were obtained using ADDR (see the section on compiler related commands).

7th byte = 0 This again tells the tracker to expect a control code.

8th byte = 0 This control code tells the tracker to execute the first instruction. Since there is only one other instruction (the 'XBNC') this means that the tracker will loop back and execute an 'XBNC' on every invocation.

Line 40 Since the XBNC command uses data in SET 4 it is required to assign the necessary values to all the relevant variables. We're going to start the track at column 15 and row 25 with sprite 46 so these and all the other variables as well as a bounce window are set up in this line and line 50.

Line 60 This line sets the column and row to launch at, sets the tracker number in KEY, tells the tracker that the first sprite to be put will be 46 and then launches the sprite with a TPUT. The "1" following the TPUT tells the tracker that the first instruction to execute has a program counter value of 1 i.e the first instruction after the 'type' byte.

This program sets up the tracking sprite and launches it to the screen, what we need to do now is to execute the track itself so lets add a line at 60 which will execute the track in a machine code loop. Don't clear the program in memory but just type:

```
70|TMOV,1000,1
```

This will execute 1000 times with flyback synchronisation. To execute just type "RUN".

Let's move on now to a second example. This time we'll move a sprite around the screen under joystick or keyboard control and set collision detection on. Clear the last example by typing "NEW" then enter the following:

```
5 ' EXAMPLE OF TRACKING SPRITES II
10 X%=0:Y%=0:¦SPN,41:¦ISPR,@X%,@X%,@X%,@Y%
20 DATA 206,50,2,8,0,0
30 ¦SET,0
40 FOR I%=Y%+5 TO Y%+10:READ X%:POKE I%,X%:NEXT I%
50 ¦COL,15:¦ROW,25:¦KEY,41:¦SP1,50:¦TPUT,1
60 A$="TMOVA#":¦ISET,@A$:¦IRUN,0
70 GOTO 70
```

Line 10 Initialises X% and Y% then assigns the start address of sprite 41 to Y%.

Line 20 This contains the data for the tracking sprite program and which has the following significance:

1st byte = 206 This is the move type byte. The first two bits select the type of movement which is 2 for exclusive-OR movement. The third bit tells the system whether or not the sprite is joystick/ keyboard controlled. In this case it is, so bit three is set so this adds 4. Bits 4 to 6 tell the system the joystick or keyrow, in this case the joystick so this adds 72 (If you do not have a joystick you could use 48 6 = up 5 = down R = left T = right). Finally bit 7 is set because collision is on which adds 128. Therefore:
Total = 2 + 4 + 72 + 128 = 206
or 2 + 4 + 48 + 128 = 182

2nd byte = 50 This is the number of the sprite we're moving.

3rd byte = 2 Size of X-increment.

4th byte = 8 Size of Y-increment.

5th byte = 0 Control code coming next.

6th byte = 0 Run from start.

Line 30 Selects set 0.

Line 40 POKEs the data into the tracking sprite.

Line 50 Launches the sprite at column 15, row 25.

Line 60 Sets the tracker running under interrupts using SET 0.

Line 70 Endless loop to prevent return to command mode. In command mode the keys codes/joystick control codes would be printed to the screen.

In our third example we're going to set up two tracking sprites, one which will move in a clockwise square and one which will move in an anticlockwise square.

```
10 ' EXAMPLE OF TRACKING SPRITES III
20  DEFINT A-Z
30  ¦SCLS
40  UP=248:DN=8:LT=254:RT=2
50  X%=0:Y%=0
60  ¦SPN,42:¦ISPR,@X%,@X%,@X%,@Y%:POKE Y%+5,2:Y%=Y%+6
70  C=0:R=0:S=52
80  C=RT:R=0:FOR N=1 TO 17:GOSUB 250:NEXT N
90  C=0:R=DN:FOR N=1 TO 9 :GOSUB 250:NEXT N
100 C=LT:R=0:FOR N=1 TO 17:GOSUB 250:NEXT N
110 C=0:R=UP:FOR N=1 TO 9 :GOSUB 250:NEXT N
120 C=0:R=0:S=0:GOSUB 250
130 ¦SPN,43:¦ISPR,@X%,@X%,@X%,@Y%:POKE Y%+5,2:Y%=Y%+6
140 C=0:R=0:S=52
150 C=0:R=DN:FOR N=1 TO 9 :GOSUB 250:NEXT N
160 C=RT:R=0:FOR N=1 TO 17:GOSUB 250:NEXT N
170 C=0:R=UP:FOR N=1 TO 9 :GOSUB 250:NEXT N
180 C=LT:R=0:FOR N=1 TO 17:GOSUB 250:NEXT N
190 C=0:R=0:S=0:GOSUB 250
```

```
191 LOCATE 1,18:FOR N=1 TO 4:PRINT"      LASER BASIC FROM OCEAN I.Q.":NEXT N
200 |SET,0:|COL,0 :|ROW,100:|KEY,42:|SP1,52:|TPUT,1
210 |TMOV,520,1
220 |SET,0:|COL,40:|ROW,100:|KEY,43:|SP1,52:|TPUT,1
230 |TMOV,520,1
240 END
250 POKE Y%,S:Y%=Y%+1:POKE Y%,C:Y%=Y%+1:POKE Y%,R:Y%=Y%+1:RETURN
```

Line 20 Declares all variables to be integer.
Line 30 Clears the screen.
Line 40 Sets up four constants which correspond to the X and Y increments for up and down, left and right. Note that 248 is -8 and 254 is -1.
Line 50 Initialises X% and Y%.
Line 60 Interrogates sprite 42 (the tracker) to find its start address then POKEs the move type byte with 2 (exclusive-OR movement). The data pointer (Y%) is then incremented to point to the first program byte.
Line 70 Initialises C (X increment), R (Y increment) and S (sprite number).
Line 80 POKEs 17 instructions into the tracker which moves it 17 characters to the right.
Line 90 POKEs 9 instructions into the tracker which moves it 9 characters downwards.
Line 100 As line 80 but movement is left.
Line 110 As line 90 but movement is upward.
Line 120 POKEs 3 bytes (only the first 2 are used) which tells the tracker to loop back to the start of the tracker program.
Line 130 This line, together with 140,150,160,170,180 and 190 do the same thing as lines 70 to 120 but this time an anticlockewise track is generated.
Line 191 Prints some text onto the screen.
Line 200 Launches the first tracker.
Line 210 Executes the whole track 10 times.
Line 220 Launches the second tracker.
Line 230 Executes the second track 10 times.
Line 250 Subroutine which POKEs the data into the tracker. The first byte is the sprite number, the second is the X-increment and the third is the Y-increment.

The final example in this section combines trackers from the previous three examples so you will need to execute those examples before attempting this one.

```
5 ' EXAMPLE OF TRACKING SPRITES IV
10  |SET,4
20  |KEY,51:|HGT,4:|LEN,2 :|COL,0 :|ROW,75:|SP1,46:|SP2,47:|SP3,48:|SP4,49
30  |SET,0
40  |SCLS
50  X%=0:Y%=0
60  |SPN,44:|ISPR,@X%,@X%,@X%,@Y%
70  FOR N=0 TO 18:READ A:POKE Y%+N,A:NEXT N
80  |SPN,45:|ISPR,@X%,@X%,@X%,@Y%:TGT=Y%+3
90  FOR N=0 TO 6 :READ A:POKE Y%+N,A:NEXT N
100 |SPN.42:|ISPR,@X%,@X%,@X%,@Y%
110 POKE Y%+162,0:POKE Y%+163,1:POKE Y%+164,43
120 |SPN.43:|ISPR,@X%,@X%,@X%,@Y%
130 POKE Y%+162,0:POKE Y%+163.1:POKE Y%+164,42
140 |SPN,44:|CPUT
150 |SPN,45
160 |CMOV,1,1:IF PEEK(TGT)<>0 THEN PRINT CHR$(7):LOCATE 1,1
170 GOTO 160
180 DATA 40,0,75.46,1,0,41,15,25,50,1,0,42,0,100,52,1,0,0
190 DATA 42,0,41,0,40,0,0
```

Line 10 Selects SET 4.
Line 20 Sets the information in SET 4 which will be used by the bouncing sprite in tracker 40.
Line 30 Selects SET 0.
Line 40 Clears the screen.
Line 50 initialises X% and Y%.
Line 60 Interrogates sprite 44 and assigns the start address to Y%.
Line 70 POKEs the 19 bytes of data that will launch the 3 trackers. There are six bytes for each sprite to be launched (see CPUT) and one delimiter.
Line 80 Interrogates sprite 45 and assigns the start address to Y%. The address of the second

sprite's collision detection address (see CMOV) is assigned to TGT.

Line 90 POKEs the 7 bytes of data that will control the 3 trackers. There are two bytes for each sprite to be controlled and a delimiter (see CMOV).

Line 100 Interrogates sprite 42 and assigns the value to Y%.

Line 110 Adjusts the end of tracker 42 so that instead of simply looping round, control jumps to the start of tracker 43.

Line 120 Interrogates sprite 43 and assigns the value to Y%.

Line 130 Adjusts the end of tracker 43 so that instead of simply looping round, control jumps to the start of tracker 42. Thus control will loop around 42 and 43.

Line 140 Launches the 3 trackers.

Line 150 Selects sprite 45 (to execute the CMOV).

Line 160 Executes the controller and then checks to see if the joystick controlled sprite collided with any other screen image, if so then a sound is made.

Line 170 Loops back to 160.

Line 180 Data for the CPUT.

Line 190 Data for the CMOV.

This concludes the section on Tracker sprites and the "GETTING STARTED" example programs.

LASER BASIC EXTENDED COMMANDS IN DETAIL

SPRITE UTILITIES

Command — Action

SSPR,e1,e2 — Sets up sprites. This command needs to be executed before any sprite operations can be carried out. The first expression, e1, tells the system what the maximum sprite number is, and hence, how much table space to allocate. The table will allocate 4 bytes for each sprite and 4 bytes for sprite 0. The second expression, e2, tells the system the address of sprite space, and the table and sprites will expand downwards from that address. Any existing sprites are destroyed.

Parameters — Use

e1 — BASIC expression which gives the maximum sprite number which the user can utilise. If a sprite number is used with a value greater than e1, then ** SPN TOO HIGH ** will be displayed. If the value of the expression is 0 then ** SPN OF ZERO ** will be displayed. In both cases, no action will be taken. In fact e1 must be in the range 1 to 255.

e2 — This BASIC expression tells the system the lowest protected byte of memory. Sprites will build down from the first byte below that address.

Example: — SSPR,7,&7000 will make the maximum sprite number 7, and the top of sprites equal to 6FFF hex.

Command — Action

DSPR — Delete the sprite whose number is held in SPN. The sprite table entry is cleared, the sprite data deleted and sprite space is contracted upwards. If the sprite was not previously defined then ** SPN DOESN'T EXIST ** will be displayed.

Parameter — Use

SPN — Number of sprite to be deleted.

Command — Action

ESPR — Increase/decrease maximum sprite number and expand/contract sprite space. If the new maximum is lower than the old maximum then all existing sprites with numbers greater than the new sprite maximum are deleted.

Parameter — Use

SPN — New maximum sprite number.

Command — Action

CSPR — Create a sprite with the number held in SPN and dimensions held in HGT and LEN. Sprite space extends down. If the sprite already exists then ** SPN EXISTS ** will be displayed.

Parameter — Use

SPN — Number of sprite to be created.
HGT — HGT in pixels of new sprite.
LEN — Width in bytes of the new sprite.

NOTE: — Each byte will be 4 pixels wide in 4 colour mode and 2 pixels wide in 16 colour mode.

Command — Action

RSPR,e1 — Relocate sprite space by the signed increment, given by the BASIC expression e1. A positive value will move sprites to higher memory, and a negative expression will move sprites to a lower address. This command will very rarely be used.

Parameters Use

e1 BASIC expression which gives the size and direction of the relocation.

Command Action

PSPR,@V$ Puts sprites to tape or disk. The filename must be held in a string variable and must be in the form "NAMESPR" . In fact three files are saved. The first file contains the system variables SMAX, STAB, SPST and SPND which contain the maximum sprite number, start of table, start of sprite data and end of sprite data respectively. If "SPR" is not found as the last 3 characters of the filename then ** ILLEGAL FILENAME ** is displayed. Filenames must be typed in upper case.

Parameters Use

@V$ The address of the 3 byte descriptor for the filename under which the three files "NAMESYS", "NAMETAB" and "NAMESPR" are to be saved.

Example: A$="TESTSPR" followed by PSPR,@A$ will create three files:
"TESTSYS" The current system variables
"TESTTAB" The sprite table
"TESTSPR" The actual sprite data

Command Action

GSPR,@V$ Gets sprites from tape or disk. The filename is again held in a string and the last 3 characters must be "SPR" or the last 7 chracters must be "SPR.BAK" . The three files are loaded and the sprites are positioned where they were saved from. Filenames must be typed in upper case. Tape prompts are suppressed.

Parameter Use

@V$ The address of the 3 byte descriptor for the filename of the files to load.

Example: A$="TESTSPR" followed by GSPR,@A$ will load the three files:
"TESTSYS", "TESTTAB" and "TESTSPR".

Command Action

MSPR,@V$ Merge sprites from tape or disk. The filename specifier is the same as that employed by PSPR and GSPR. The table of the file being loaded is merged with the resident file. If however, a sprite being loaded has an existing sprite number, then an error is generated and no further action is taken. If this happens the filename will be of the form "NAMETAB" and will need to be reassigned before a second attempt.

If the tables are successfully merged and the highest sprite number being loaded is less than or equal to the current maximum sprite number then the sprite data is loaded and sprite space expanded downward to accomodate the new sprites. MSPR can be used to effectively load sprites into an address other than that from which they were saved. Filenames must be typed in upper case. Tape prompts are suppressed.

Parameter Use

@V$ The address of the 3 byte file descriptor for the sprite to be merged.

Note: If the ** OUT OF MEMORY ** error is generated you will need to re-set MEMORY (Amstrad BASIC) and perform an MSET, to a lower address. This applies to GSPR and MSPR.

Command Action

RNUM Renumber sprite SP1 (if it exists) to become sprite SP2 (if it doesn't already exist). SP1 ceases to exist.

Parameters Use

SP1 Sprite to be renumbered.
SP2 New sprite number.

Command Action

ADNM Increment all existing sprite numbers by the value held in SPN. Errors will be generated if this would cause a current sprite to exceed the maximum sprite number. The value in SPN must be positive and non-zero.

Parameter Use

SPN Holds the value by which all current sprite numbers are to be incremented.

Command Action

ISPR,@**V1**,@**V2**,@**V3**,@**V4** Interrogate sprite detaiis. The following variables are assigned the respective system information:

V1 Start of sprite table (lowest address utilised by sprites).
V2 Start of sprite data.
V3 End of sprites (highest address utilised by sprites).
V4 Address of the data of the sprite whose number is held in SPN.

In addition to the above, HGT and LEN are set to the dimensions of the sprite whose number is held in SPN. If the sprite is found not to exist then no error message is generated, but instead SPN is set to zero and the BASIC and grapnics variables are left unchanged.

Command Action

MASK Create a masked sprite from an unmasked sprite. The move commands FMOV and BMOV will allow sprites to move non-destructively in Front of, or Behind, screen data. This is not an exclusive-OR operation but instead provides the user with a facility similar to that afforded by hardware sprites. The technique employed involves generating a negative mask of pixel data and creating a sprite with alternating data and mask bytes. This means that the width of the displayed sprite is half the physical size of the sprite. Thus any sprite to be masked must have an even physical width or ** CAN'T MASK ** will be displayed. Prior to masking, the data to be displayed, should occupy only the left hand half of the sprite.

Masked sprites should only be used with the commands FMOV, BMOV, FMVJ, BMVJ, FBNC, BBNC, FSWP, BPUT, BGET, RMSK, DMSK, MASK or utilised as tracking sprites.

Parameter Use

SPN The number of the sprite to be masked.

Command Action

RMSK Re-create a masked sprite from a sprite which has previously been masked but needs re-masking. This would be the case if a sprite had been moved behind screen data and was then required to move in front of screen data.

Command Action

DMSK De-mask a previously masked sprite and re-create the pixel data in the left hand half. The right hand half will be cleared.

Parameter Use

SPN The number of the sprite to be de-masked.

Command Action

HRSP Create hi-res sprite pair. When the software is put into 160 column mode, the column resolution can be effectively halved. This is achieved by dividing the column value by 2 and adding the remainder (1 or 0) to the sprite number and this is carried out automatically after execution of CLHI and continues until the system is returned to 80 column mode using CLLO. The action of HRSP then, is to produce a new sprite, with sprite number, 1 greater than the specified sprite. This new sprite is identical to the specified sprite but the data is scrolled by a half byte (2 pixels in 4 colour mode or 1 pixel in 16 colour mode). If sprite SPN+1 already exists then an error results.

IMPORTANT NOTE:

MASK, RMSK, DMSK and HRSP test the mode flag before execution. If the operation is to be carried out on 4 colour data then ONHI MUST be executed before any of the former. If the operation is to be carried out on 16 colour data then ONLO MUST be executed before any of the former.

Parameter — Use

SPN — The number of the sprite to be paired.

NOTE: — The sprite to be paired should not contain any data in the rightmost half column as this will be wrapped into the leftmost half column of the target sprite.

Command — Action

FREE,@V1 — Test the amount of free sprite space and assign the result to V1. This command is provided to interrogate the amount of free space available for sprites, and returns the result into the variable V1.

Parameter — Use

V1 — The amount of free sprite space is returned to the BASIC integer variable V1.

Command — Action

MSET,e1 — Set MBOT. This tells the system the lowest free byte available for sprites and workspace and should be set to HIMEM + 1 each time BASIC's MEMORY command is executed.

Parameter — Use

e1 — The value of the expression e1 is assigned to the system variable e1.

PARAMETER RELATED COMMANDS

Command — Action

SET,e1 — The value of the expression e1 is assigned to the graphics variable SET. The value for the expression must be in the range 0 to 15 and is used to select one of the 16 sets of graphics variables.

XCL,e1 — The value of the expression e1 is assigned to the graphics variable XCL. The value must be in the range 0 to 319 and is used by the FILL command to give the X-coordinate of the point at which to begin FILLing.

IK1,e1 — The value of the expression e1 is assigned to the graphics variable IK1. The value must be in the range 0 to 15 and is used to set the INK number for the FILL, STCV and SETV commands.

IK2,e1 — The value of the expression e1 is assigned to the graphics variable IK2. The value must be in the range 0 to 15 and is used to set the INK number for the SETV command.

COL,e1 — The value of the expression e1 is assigned to the graphics variable COL. The value must be in the range -128 to 255 and is used to define the screen column for various operations.

ROW,e1 — The value of the expression e1 is assigned to the graphics variable ROW. The value must be in the range -128 to 255 and is used to define the screen row (in pixels) for operations.

LEN,e1 — The value of the expression e1 is assigned to the graphics variable LEN. The value must be in the range -128 to 255 and is used variously to define the width of screen windows, sprite windows, sprite dimensions and the X-increment for the move commands.

HGT,e1 — The value of the expression e1 is assigned to the graphics variable HGT. The value must be in the range -128 to 255 and is used in similar applications to LEN.

SPN,e1 The value of the expression e1 is assigned to the graphics variable SPN. The value must be in the range 1 to 255 and is used to specify the sprite number in a variety of applications.

SP1,e1 The value of the expression e1 is assigned to the graphics variable SP1. The value must be in the range 1 to 255 and is used to specify one of two sprite numbers in sprite to sprite operations or one of four sprite numbers in animated sequences.

SP2,e1 The value of the expression e1 is assigned to the graphics variable SP2. The value must be in the range 1 to 255 and is used in a similar manner to SP1 above.

SP3,e1 The value of the expression e1 is assigned to the graphics variable SP3. The value must be in the range 1 to 255 and is used as one of four sprite numbers in an animated sequence.

SP4,e1 The value of the expression e1 is assigned to the graphics variable SP4. The value must be in the range 1 to 255 and is used as one of the four sprite numbers in an animated sequence.

SCL,e1 The value of the expression e1 is assigned to the graphics variable SCL. The value must be in the range 0 to 255 and is used to specify the column of a sprite window within a sprite or the screen column for the target of a rotation.

SRW,e1 The value of the expression e1 is assigned to the graphics variable SRW. The value must be in the range 0 to 255 and is used to specify the row of a sprite window within a sprite or the screen row for the target of a rotation.

NPX,e1 The value of the expression e1 is assigned to the graphics variable NPX. The value must be in the range -128 to 255 and is used to specify the size (in pixels) and direction by which to vertically scroll a screen window, sprite window or whole sprite.

KEY,e1 The value of the expression e1 is assigned to the graphics variable KEY. The value must be in the range 0 to 79 and specifies amongst other things the KEY to be scanned for by the KBFN command or the key/joystick row to be scanned by the FMVJ, BMVJ, XMVJ and WMVJ commands.

SETQ,@V1 The value in the graphics variable SET is assigned to the BASIC variable V1. V1 must be an integer variable.

XCLQ,@V1 The value in the graphics variable XCL is assigned to the BASIC variable V1. V1 must be an integer variable.

IK1Q,@V1 The value in the graphics variable IK1 is assigned to the BASIC variable V1. V1 must be an integer variable.

IK2Q,@V1 The value in the graphics variable IK2 is assigned to the BASIC variable V1. V1 must be an integer variable.

COLQ,@V1 The value in the graphics variable COL is assigned to the BASIC variable V1. V1 must be an integer variable.

ROWQ,@V1 The value in the graphics variable ROW is assigned to the BASIC variable V1. V1 must be an integer variable.

LENQ,@V1 The value in the graphics variable LEN is assigned to the BASIC variable V1. V1 must be an integer variable.

HGTQ,@V1 The value in the graphics variable HGT is assigned to the BASIC variable V1. V1 must be an integer variable.

SPNQ,@V1 The value in the graphics variable SPN is assigned to the BASIC variable V1. V1 must be an integer variable.

SP1Q,@V1 The value in the graphics variable SP1 is assigned to the BASIC variable V1. V1 must be an integer variable.

SP2Q,@V1 The value in the graphics variable SP2 is assigned to the BASIC variable V1. V1 must be an integer variable.

SP3Q,@V1 The value in the graphics variable SP3 is assigned to the BASIC variable V1. V1 must be an integer variable.

SP4Q,@V1 The value in the graphics variable SP4 is assigned to the BASIC variable V1. V1 must be an integer variable.

SCLQ,@V1 The value in the graphics variable SCL is assigned to the BASIC variable V1. V1 must be an integer variable.

SRWQ,@V1 The value in the graphics variable SRW is assigned to the BASIC variable V1. V1 must be an integer variable.

NPXQ,@V1 The value in the graphics variable NPX is assigned to the BASIC variable V1. V1 must be an integer variable.

KEYQ,@V1 The value in the graphics variable KEY is assigned to the BASIC variable V1. V1 must be an integer variable.

EXXV Exchange foreground and background SET variables. In order to allow the user to utilise locomotive BASIC's powerful EVERY and AFTER commands it is necessary to be able to save the current value for SET and assign the new value that the interrupt now requires. EXXV should be executed as the first and last commands of all interrupt routines.

SWPS Swop the sprite numbers in SP2 and SP4. This command is provided so that the order of an animation sequence can be reversed.

ASTV Assigns sprite data to current variable SET. The sprite should contain 20 bytes of information and must have a height of 1 and a length of 20, otherwise a parameter error will be issued.

Parameter	Use
SPN	Number of the sprite containing the data for variables. SET The set to which the data should be assigned.

AVTS Assign the current variable set to the sprite whose number is held in SPN. The sprite must have a height of 1 and a length of 20, otherwise a parameter error will be issued.

Parameter	Use
SPN	Number of the sprite which will contain data.
SET	The set of variables to be assigned.

ESAV Exchange the current variable set with the data in the sprite whose number is held in SPN. The sprite must have a height of 1 and a length of 20, otherwise a parameter error will be issued.

Parameter	Use
SPN	Number of the sprite containing the data to be exchanged.
SET	The set of variables to be exchanged.

SYSTEM SWITCHES

ONLO Puts the hardware and software into 16 colour mode. This should be executed instead of BASIC's MODE command.

ONHI Puts the hardware and software into 4 colour mode. Again this replaces the use of BASIC's MODE command.

CLLO Puts the software into 80 column mode. In this mode column 79 is the rightmost column.

CLHI Puts the software into 160 column mode. In this mode, all GETs and PUTs treat the screen as being 160 columns wide and column 159 is the rightmost column. Note, however, that for all other purposes the screen is still treated as having 80 columns and in particular, windows defined for scrolling, mirroring etc., will still treat the screen as having 80 columns.

IMPORTANT NOTE:

When operations are carried out on sprites or sprite windows the ONHI and ONLO commands are needed to inform the system that the mode has been changed BEFORE the operation is carried out. If, for instance, a 16 colour sprite is MASKed and the last switch used was ONHI, then the sprite will be corrupted.

GROUP 1 GETs and PUTs

Group 1 GETs and PUTs are prefixed with GT and PT respectively. GET commands use a screen window as their source of data and a sprite as their destination. PUTs use a sprite as their source and the screen as their destination. Each command has one of six suffixes.

BL Data is block moved from source to destination and replaces the data previously held in the destination.

OR Data from the source is ORed into the data currently held in the destination.

ND Data from the source is ANDed into the data currently held in the destination.

XR Data from the source is XORed into the data currently held in the destination.

BH Data from the source is placed behind data in the destination.

IF Data from the source is placed in front of data in the destination.

In each case, SPN is used to specify the sprite and COL and ROW are used to define the top left hand corner of a screen window. The dimensions of the window are the dimensions of the sprite and in the event of the window overlapping the screen border, the operation takes place on the "on-screen" portion of the window. If the sprite is wholly off screen no operation takes place but no errors are generated.

For each of the twelve commands in Group 1:

Parameter	Use
SPN	The number of the sprite to be used.
COL	The screen column of the window.
ROW	The screen row of the window.
@V1	Used in collision detection (see next section).

Command	Action
GTBL	Block move screen window into sprite.
GTOR	OR screen window into sprite.
GTND	AND screen window into sprite.
GTXR	XOR screen window into sprite.
GTBH	Place screen window data behind sprite data.
GTIF	Place screen window data in front of sprite data.
PTBL	Block move sprite into screen window.
PTOR	OR sprite into screen window.
PTND	AND sprite into screen window.
PTXR	XOR sprite into screen window.
PTBH	Place sprite data behind screen window data.
PTIF	Place sprite data in front of screen window data.

Collision Detection

Each of the 12 commands in this group can be executed with or without collision detection. If any of the commands in the group is executed without a following parameter then detection is flagged "off". If the command is executed with the address of an integer variable as its single operand, then detection is flagged "on". Collision detection will slow the execution and should only be used when necessary. If the source data collides with the target data then the value in the BASIC variable is incremented, but if not, remains unchanged.

Example: If the value of X% was set to 7 and the command PTBH,@X% were executed, then collision between the sprite and screen data would cause X% to be incremented so that it subsequently held 8.

GROUP II GETs and PUTs

Group II GETs and PUTs are prefixed with GW and PW respectively. The GET commands use a screen window as their source of data and a sprite window as their destination. The PUTs use a sprite window as their source and a screen window as their destination. Each command has one of six suffixes:

BL Data is block moved from source to destination and replaces the data previously held in the destination.

OR Data from the source is ORed into the data currently held in the destination.

ND Data from the source is ANDed into the data currently held in the destination.

XR Data from the source is XORed into the data currently held in the destination.

BH Data from the source is placed behind data in the destination.

IF Data from the source is placed in front of data in the destination.

In each case SPN is used to specify the sprite, SCL and SRW specify the column and row of the sprite window within the sprite, COL and ROW specify the column and row of the screen window and HGT and LEN specify the dimensions of the window. The dimensions of the window will be reduced if the window lies partially "off-sprite" or "off-screen". If the window is wholly "off-sprite" or "off-screen" then no operation will take place, but no error message will be generated.

For each of the twelve commands in Group II:

Parameter	Use
SPN	The number of the sprite to be used.
SCL	The sprite column of the window.
SRW	The sprite row of the window.
COL	The screen column of the window.
ROW	The screen row of the window.
LEN	The width of the window.
HGT	The height of the window.
@V1	Used in collision detection.

Command	Action
GWBL	Block move screen window into sprite window.
GWOR	OR screen window into sprite window.
GWND	AND screen window into sprite window.
GWXR	XOR screen window into sprite window.
GWBH	Place screen window data behind sprite window data.
GWIF	Place screen window data in front of sprite window data.
PWBL	Block move sprite window into screen window.
PWOR	OR sprite window into screen window.
PWND	AND sprite window into screen window.
PWXR	XOR sprite window into screen window.
PWBH	Place sprite window data behind screen window data.
PWIF	Place sprite window data in front of screen window data.

Collision Detection

Collision detection for the Group II commands works in exactly the same way as the previously described Group I commands. If no parameter follows the command then detection is off, if an integer variable address follows the command then the variable will be incremented if collision takes place, or remain unchanged if no collision is detected. Again detection will slow the operation of the command.

GROUP III GETs and PUTs

Group III GETs and PUTs are prefixed with GM and PM respectively. The GET commands use a sprite window as their source of data and a whole sprite as their destination. The PUTs use a whole sprite as their source and a sprite window as their destination. Each command has one of six suffixes:

BL Data is block moved from source to destination and replaces the data previously held in the destination.

OR Data from the source is ORed into the data currently held in the destination.

ND Data from the source is ANDed into the data currently held in the destination.

XR Data from the source is XORed into the data currently held in the destination.

BH Data from the source is placed behind data in the destination.

IF Data from the source is placed in front of data in the destination.

In each case SP1 is used to specify the whole sprite, SP2 specifies the sprite containing the window, SCL and SRW specify column and row of the window in sprite SP2 and dimensions of the window are the dimensions of sprite SP1. The dimensions of the window will be reduced of the values of SCL and SRW cause sprite SP1 to overlap the borders of sprite SP2. If SP1 lies wholly "off-sprite" then no operation will take place but no error will be generated.

For each of the twelve commands in Group III:

Parameter	Use
SP1	The number of the whole sprite.
SP2	The number of the sprite containing the sprite window.
SCL	Column of the sprite window in sprite SP2.
SRW	Row of the sprite window in sprite SP2.
@V1	Used in collision detection.

Command	Action
GMBL	Block move sprite window into whole sprite.
GMOR	OR sprite window into whole sprite.
GMND	AND sprite window into whole sprite.
GMXR	XOR sprite window into whole sprite.
GMBH	Place whole sprite data behind sprite window data.
GMIF	Place whole sprite data in front of sprite window data.
PMBL	Block move whole sprite into sprite window.
PMOR	OR whole sprite into sprite window.
PMND	AND whole sprite into sprite window.
PMXR	XOR whole sprite into sprite window.
PMBH	Place sprite window data behind whole sprite data.
PMIF	Place sprite window data in front of whole sprite data.

Collision Detection

Collision detection for the Group III commands works in exactly the same way as the previously described Group I and II commands. If no parameter follows the command then detection is off, if an integer variable address follows the command then the variable will be incremented if collision takes place or remain unchanged if no collision is detected. Again, detection will slow the operation of the command.

IMPORTANT NOTE:

Ensure that ONLO and ONHI are used to put Laser BASIC into the correct mode before using 'GM' or 'PM' commands.

GROUP I SCROLLS AND WRAPS

Group I scrolls and wraps are prefixed with SV and WV respectively. The scroll commands scroll data without wrap and the wraps scroll data with wrap. Each command has one of six suffixes:

R1 Data moves right by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

L1 Data moves left by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

R4 Data moves right by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

L4 Data moves left by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

R8 Data moves right by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

L8 Data moves left by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

In each case COL and ROW specify the column and row of the top left of the screen window and HGT and LEN specify the dimensions of the window. The dimensions of the window will be adjusted if the window lies partially "off-screen". If the window is wholly "off-screen" then no operation will take place but no error will be generated.

For each of the twelve commands in Group I:

Parameter	Use
COL	Column of screen window.
ROW	Row of screen window.
LEN	Width of screen window.
HGT	Height of screen window.
e1	Optional parameter to specify number of executions.
e2	Optional parameter to specify frame sync status.

Command	Action		
SVR1	Scroll screen window right,	1 pixel,	no wrap.
SVL1	Scroll screen window left,	1 pixel,	no wrap.
SVR4	Scroll screen window right,	1 byte,	no wrap.
SVL4	Scroll screen window left,	1 byte,	no wrap.
SVR8	Scroll screen window right,	2 bytes,	no wrap.
SVL8	Scroll screen window left,	2 bytes,	no wrap.
WVR1	Scroll screen window right,	1 pixel,	with wrap.
WVL1	Scroll screen window left,	1 pixel,	with wrap.
WVR4	Scroll screen window right,	1 byte,	with wrap.
WVL4	Scroll screen window left,	1 byte,	with wrap.
WVR8	Scroll screen window right,	2 bytes,	with wrap.
WVL8	Scroll screen window left,	2 bytes,	with wrap.

Repeated Execution

If any of the Group I scrolls or wraps (or Group II or III scrolls or wraps for that matter) is executed without any following parameters, then the command will simply execute once. If, however, the command is followed by 2 expressions then the command will repetetively execute. The first expression sets the number of times the command will execute. The second parameter specifies whether the command will synchronise with frame-flyback or not. A zero value for the second parameter causes the command to repeatedly execute without delay but a non-zero value causes the execution to commence 50 times a second (or multiples of 1/50 second period depending on the execution time) and thus produces much smoother movement. If the command is followed by 1 parameter or more than 2 parameters then ** PARAMETER ERROR ** will be displayed.

GROUP II SCROLLS AND WRAPS

Group II scrolls and wraps are prefixed with SS and WS respectively. The scroll commands scroll data without wrap and the wraps scroll data with wrap. Each command has one of six suffixes:

R1 Data moves right by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

L1 Data moves left by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

R4 Data moves right by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

L4 Data moves left by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

R8 Data moves right by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

L8 Data moves left by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

In each case, SPN specifies the sprite to be scrolled or wrapped, so for each of the twelve commands in Group I:

Parameter	Use
SPN	Sprite to be scrolled or wrapped.

Command	Action		
SSR1	Scroll sprite right,	1 pixel,	no wrap.
SSL1	Scroll sprite left,	1 pixel,	no wrap.
SSR4	Scroll sprite right,	1 byte,	no wrap.
SSL4	Scroll sprite left,	1 byte,	no wrap.
SSR8	Scroll sprite right,	2 bytes,	no wrap.
SSL8	Scroll sprite left,	2 bytes,	no wrap.
WSR1	Scroll sprite right,	1 pixel,	with wrap.
WSL1	Scroll sprite left,	1 pixel,	with wrap.
WSR4	Scroll sprite right,	1 byte,	with wrap.
WSL4	Scroll sprite left,	1 byte,	with wrap.
WSR8	Scroll sprite right,	2 bytes,	with wrap.
WSL8	Scroll sprite left,	2 bytes,	with wrap.

Repeated Execution

Group II scrolls and wraps can be repeatedly executed in exactly the same manner as previously described for Group I repeated execution.

GROUP III SCROLLS AND WRAPS

Group III scrolls and wraps are prefixed with SP and WP respectively. The scroll commands scroll data without wrap and the wraps scroll data with wrap. Each command has one of six suffixes:

R1 Data moves right by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

L1 Data moves left by 1 pixel. This is 1/320 of the screen width in 4 colour mode or 1/160 of the screen width in 16 colour mode.

R4 Data moves right by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

L4 Data moves left by 1 byte. This is 1/80 of the screen width in both modes. In 4 colour mode this represents a 4 pixel move and in 16 colour mode this represents a 2 pixel move.

R8 Data moves right by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

L8 Data moves left by 2 bytes. This is 1/40 of the screen width in both modes. In 4 colour mode this represents an 8 pixel move and in 16 colour mode this represents a 4 pixel move.

In each case SPN specifies the sprite containing the window to be scrolled, COL and ROW contain the column and row at the top left of the window within the sprite, and LEN and HGT specify the dimensions of the window. If the choice of COL and ROW causes the window to lie partially "off-sprite" then the window dimensions are adjusted accordingly and if the window lies wholly "off-sprite" then no execution takes place but no error message is generated.

Parameter	Use
SPN	Number of the sprite containing the window.
COL	Sprite column of the window.
ROW	Sprite row of the window.
LEN	Width of the window.
HGT	Height of the window.

Command	Action		
SPR1	Scroll sprite window right,	1 pixel,	no wrap.
SPL1	Scroll sprite window left,	1 pixel,	no wrap.
SPR4	Scroll sprite window right,	1 byte,	no wrap.
SPL4	Scroll sprite window left,	1 byte,	no wrap.
SPR8	Scroll sprite window right,	2 bytes,	no wrap.
SPL8	Scroll sprite window left,	2 bytes,	no wrap.
WPR1	Scroll sprite window right,	1 pixel,	with wrap.
WPL1	Scroll sprite window left,	1 pixel,	with wrap.
WPR4	Scroll sprite window right,	1 byte,	with wrap.
WPL4	Scroll sprite window left,	1 byte,	with wrap.
WPR8	Scroll sprite window right,	2 bytes,	with wrap.
WPL8	Scroll sprite window left,	2 bytes,	with wrap.

Repeated Execution

Group III scrolls and wraps can be repeatedly executed in exactly the same manner as previously described for Group I and II repeated execution.

GROUP IV SCROLLS AND WRAPS

These are the vertical scrolls and wraps, and each is suffixed with VN (Vertically NPX). The scroll commands scroll without wrap and are prefixed with S, whilst the wraps scroll with wrap, and are prefixed with W. There are three different types (given by the second character in the name) and these are:

Type V

Scrolling takes place on a screen window, where COL and ROW specify the screen position of the window top left and HGT and LEN specify the window dimensions. The window parameters are adjusted if it lies partially "off-screen".

Type S

Scrolling takes place on the whole sprite whose number is held in SPN.

Type P

Scrolling takes place on a sprite window. The number of the sprite is given by SPN and the window is specified by COL, ROW, HGT and LEN. The window is adjusted if it lies partially "off-sprite".

Command	Action
SVVN	The screen window is scrolled vertically by NPX pixels, without wrap.
WVVN	The screen window is scrolled vertically by NPX pixels, with wrap.

Parameter	Use
COL	Screen column of window top left.
ROW	Screen row of window top left.
LEN	Width of screen window.
HGT	Height of screen window.
NPX	Size and direction of scroll in pixels (1/200 of screen height). Positive values cause upward movement and negative values cause downward movement.

Command	Action
SSVN	The sprite is scrolled vertically by NPX pixels, without wrap.
WSVN	The sprite is scrolled vertically by NPX pixels, with wrap.

Parameter	Use
SPN	Number of the sprite to be scrolled.
NPX	Size and direction of scroll in pixels.

Command	Action
SPVN	The sprite window is scrolled vertically by NPX pixels, without wrap.
WPVN	The sprite window is scrolled vertically by NPX pixels, with wrap.

Parameters	Use
SPN	Number of sprite to be scrolled.
COL	Sprite column of window top left.
ROW	Sprite row of window top left.
LEN	Width of sprite window.
HGT	Height of sprite window.
NPX	Size and direction of scroll in pixels.

Repeated Execution

Group IV scrolls and wraps can be repeatedly executed in exactly the same manner as previously described in Group I, II and III commands.

IMPORTANT NOTE:

Ensure that ONLO and ONHI have been executed correctly before carrying out any operations on sprites or sprite windows.

TRANSFORMATIONS

A range of commands are included in the package which carry out various transformations on sprites, sprite windows or screen windows. In each case only the window or the sprite itself is affected, and there is no flow of data between the window or sprite being transformed and any other windows or sprites. There are three data types (indicated by the command suffix) and these are:

Type V

Transformations take place on a screen window where COL and ROW specify the screen position of the window top left and HGT and LEN specify the window dimensions. The window parameters are adjusted if it lies partially "off-screen".

Type S

The transformation takes place on the whole sprite whose number is held in SPN.

Type P

The transformation takes place on a sprite window. The number of the sprite is given by SPN and the window is specified by COL, ROW, HGT and LEN. The window is adjusted if it lies partially "off-sprite".

GROUP I TRANSFORMATIONS

Group I transformations are all type V and are carried out on screen windows. Each command is suffixed with V. For the following commands in Group I:

Parameter	Use
COL	Screen column of window top left.
ROW	Screen row of window top left.
LEN	Window width.
HGT	Window height.

Command	Action
CLSV	Clear window to paper colour (INK 0).
MGXV	X-expand left hand half of screen window into full screen window.
MGYV	Y-expand top half of screen window into full screen window.
MIRV	Mirror screen window about its vertical centre.
MORV	Mirror left hand half of screen window into right hand half (creates a horizontally symetric window).
FIPV	Mirror screen window about its horizontal centre.
FOPV	Mirror top half of screen window into bottom half (create a vertically symetric window).
INVV	Invert (1's compliment) all the pixel data in the screen window.

For the following commands in Group I:

Parameter	Use
COL	Screen column of window top left.
ROW	Screen row of window top left.
LEN	Width of window.
HGT	Height of window.
IK1	First INK colour.
IK2	Second INK colour.

Command	Action
STCV	Fill screen window with the INK whose number is held in IK1.
SETV	Re-colour a window graphic. Replaces every pixel which currently has the colour whose number is held in IK1, by a pixel which has the colour whose number is held in IK2.

GROUP II TRANSFORMATIONS

Group II transformations are all type S and are carried out on whole sprites, and each command is suffixed with S.

For the following commands in Group II:

Parameter	Use
SPN	Number of sprite to transform.

Command	Action
CLSS	Clear sprite to paper colour (INK 0).
MGXS	X-expand left hand half of sprite into full sprite.
MGYS	Y-expand top half of sprite into full sprite.
MIRS	Mirror sprite about its vertical centre.
MORS	Mirror left hand half of sprite into right hand half (creates a horizontally symetric sprite).
FIPS	Mirror sprite about its horizontal centre.
FOPS	Mirror top half of sprite into bottom half (creates a vertically symetric sprite).

For the following commands in Group II:

Parameter	Use
SPN	Number of sprite to be transformed.
IK1	First INK number.
IK2	Second INK number.

Command	Action
STCS	Fill whole sprite with the INK whose number is held in IK1.
SETS	Recolour a sprite graphic. Replaces every pixel which currently has the colour whose number is held in IK1, by a pixel which has the colour whose number is held in IK2.

GROUP III TRANSFORMATIONS

Group III transformations are all type P and are carried out on a sprite window. Each command is suffixed with P.

For the following commands in Group III:

Parameter	Use
SPN	Number of sprite to be transformed.
COL	Sprite column of the window top left.
ROW	Sprite row of window top left.
LEN	Width of sprite window.
HGT	Height of sprite window.

Command	Action
CLSP	Clear sprite window to paper colour (INK 0).
MGXP	X-expand left hand half of sprite window into full window.
MIRP	Mirror sprite window about its vertical centre.
MORP	Mirror left hand half of sprite window into right hand half (creates a horizontally symetric sprite window).
INVP	Invert (1's compliment) all the pixels in the sprite window.

For the following commands in Group III:

Parameter	Use
SPN	Number of the sprite to be transformed.
COL	Sprite column of window top left.
ROW	Sprite row of window top left.
LEN	Width of sprite window.
HGT	Height of sprite window.
IK1	First INK number.
IK2	Second INK number.

Command	Action
STCP	Fill sprite window with the INK whose number is held in IK1.
SETP	Re-colour a sprite graphic. Replaces every pixel which currently has the colour whose number is held in IK1, by a pixel which has the colour whose number is held in IK2.

IMPORTANT NOTE:

Ensure that ONLO and ONHI have been correctly executed before carrying out any operations on sprites or sprite windows.

DATA EXCHANGES

Three data exchange commands are provided which utilise MASKed sprites to allow data to be PUT behind or in front of screen data, and which store the screen data in the sprite so that the screen and sprite can be returned to their former state.

In each case, SPN is used to specify the number of the MASKed sprite and COL and ROW are used to define the top left hand corner of a screen window. The height of the window is the height of the sprite but the width of the window is only half the width of the sprite. This is because of the internal format of MASKed sprites (see MASK). If any of these commands is executed with a sprite which has not been masked then the outcome is unpredictable. If the window overlaps the screen border, the operation takes place in the "on-screen" portion of the window. If the window is wholly "off-screen" then no operation takes place but no error is generated. Although the above description may sound rather confusing, the effect of the operation is quite simple.

For each of the three commands:

Parameter	Use
SPN	The number of the masked sprite to use.
COL	The screen column of the window.
ROW	The screen row of the window.
@V1	Used in collision detection.

Command	Action
FSWP	The masked sprite is placed in front of the screen data and the screen data is lifted into the sprite. Note that executing a second FSWP at the same COL and ROW with the same masked sprite will return both the sprite and the screen to their former states. If FSWP is not executed a second time, the sprite will be left corrupted.
BPUT	The masked sprite is placed behind the screen data and the screen data is lifted into the sprite. This command does not do an exchange in the same way as FSWP does and so BPUT should not be executed twice to restore the data. Instead, data is restored using a BGET.
BGET	The masked sprite data that was placed behind the screen data by the BPUT command is lifted back up into the sprite and the screen is restored to the state it was in before the BPUT. Note that BPUT should always be followed by BGET or the masked sprite will be left in a corrupted state.
NOTE:	If a sprite is used in a BPUT-BGET sequence, then it cannot be used in a FSWP-FSWP sequence unless the sprite is re-masked using RMSK.

Collision Detection

Data exchanges support collision detection in exactly the same manner as the GETs and PUTs previously described.

LINEAR MOVE COMMANDS

There are four MOVE commands which allow a previously placed sprite to be moved from one screen position to another. The only difference between them is the type of operation they use to make the movement. All MOVE commands can be repeatedly executed and have a collision detection option. All MOVE commands are suffixed with MOV.

Animation Sequences

The MOVE commands also provide a facility for animated movement and use the four graphics variables SP1, SP2, SP3 and SP4 for this purpose. Each time a move command is executed, a check is first made to ensure that either HGT or LEN are non-zero. If both are in fact zero then no execution takes place, nor is the animation sequence effected. If one or both of HGT and LEN are non-zero then a movement will take place. At this stage it is assumed that SP1 is the number of the sprite which was previously PUT at the co-ordinates currently held in COL and ROW. This sprite is removed, the increments HGT and LEN are added to COL and ROW, and sprite SP2 is PUT at the new COL and ROW position. In addition to this, the old values for COL and ROW are replaced by the new values (incremented values) and the four sprite numbers are rotated. That is to say that SP1 takes the value SP2 held, SP2 takes the value SP3 held, SP3 takes the value SP4 held, and SP4 takes the value SP1 held. If a sprite is to be moved without animation then simply set SP1, SP2, SP3 and SP4 to contain the same sprite number.

For each of the four MOVE commands:

Parameter	Use
SP1	Sprite to be removed.
SP2	New sprite to be PUT.
SP3	Sprite to be PUT when SP2 is removed.
SP4	Sprite to be PUT when SP3 is removed.
COL	Screen column of sprite to be moved.
ROW	Screen row of sprite to be moved.
HGT	Y-increment of movement.
LEN	X-increment of movement.
@V1	Variable address for collision detection.
e1	Number of times to execute the command.
e2	Frame-flyback synchronisation flag.

Command	Action
FMOV	Move and animate, in front of screen data. Note that before the first execution of FMOV, the masked sprite needs to be PUT to the screen ready to be moved. This can be achieved either by using FSWP (when SPN is the same as SP1) or by moving from an "off-screen" position to an "on-screen" or partially "on-screen" position. An FMOV sequence should be terminated either by executing another FSWP with SPN equal to SP1, or by moving "off-screen".
FMOV,@V1	Move and animate as for FMOV, but this time collision detection is on, and the integer variable V1 is incremented if collision is detected.
FMOV,e1,e2	Move and animate as for FMOV, e1 times, with flyback synchronisation if e2 is non-zero, without flyback synchronisation if e2 is zero.
FMOV,@V1,e1,e2	Move and animate as for FMOV, with detection, e1 times, with or without flyback synchronisation.
BMOV	Move and animate, behind screen data. Note that before the first execution of BMOV, the masked sprite needs to be PUT to the screen ready to be moved. This can be achieved either by using BPUT (where SPN is the same as SP1) or by moving from an "off-screen" position to an "on-screen" or partially "on-screen" position. A BMOV sequence should be terminated either by executing a BGET with SPN equal to SP1, or by moving "off-screen". BMOV and BGET can only be used with masked sprites and every sprite which has been "BMOVed" will need to be re-masked before it can be "FMOVed".
BMOV,@V1	Move and animate as for BMOV, but this time collision detection is on and the integer variable V1 is incremented if collision is detected.
BMOV,e1,e2	Move and animate as for BMOV, e1 times, with flyback synchronisation if e2 is non-zero, without flyback synchronisation if e2 is zero.
BMOV,@V1,e1,e2	Move and animate as for BMOV, with detection, e1 times, with or without flyback synchronisation.

XMOV Move and animate, using the exclusive-OR operation. Note that before the first execution of XMOV, the non-masked sprite needs to be PUT to the screen ready to be moved. This can be achieved either by using PTXR (where SPN is the same as SP1) or by moving from an "off-screen" position to an "on-screen" or partially "on-screen" position. An XMOV sequence need not be terminated as the sprites being used are not affected by the XOR operation. If, however, the sprite is to be removed from the screen (without clearing the screen window) then this can be achieved by executing another PTXR (with SPN equal to SP1), or by simply moving "off-screen".

XMOV,@V1 Move and animate as for XMOV, but this time collision detection is on and the integer variable V1 is incremented if a collision is detected.

XMOV,e1,e2 Move and animate as for XMOV, e1 times, with flyback synchronisation if e2 is non-zero, without flyback synchronisation if e2 is zero.

XMOV,@V1,e1,e2 Move and animate as for XMOV, with detection, e1 times, with or without flyback synchronisation.

WMOV Move and animate, using a block over-write operation. The block over-write operation is a lot less sophisticated than the previous three MOVE types, but it can be as much as ten times faster in its execution. When WMOV is executed it is assumed that SP1 was PUT to the screen at the current COL and ROW. All WMOV actually does is to add LEN and HGT to the current COL and ROW values and PUT SP2 to these new co-ordinates. For this reason SP2 must have a blank border (containing no data) which is at least as large as the increments HGT and LEN, so that the old sprite SP1 is completely blotted out by the placing of the new sprite SP2. Clearly this would destroy any other screen data in the window being used by SP2, but when a sprite is being moved over a blank background with no other data present, this operation should always be used because of the enormous time savings. The sprite can be removed from the screen using CLSV or moving "off-screen".

WMOV,@V1 Move and animate as for WMOV, but this time collision detection is on, and the integer variable V1 is incremented if collision is detected. In fact there is little point in using this option, as a detection is inevitable (SP1 is being blotted out and detection does slow execution down considerably.

WMOV,e1,e2 Move and animate as for WMOV, e1 times, with flyback synchronisation if e2 is non-zero, without flyback synchronisation if e1 is zero.

WMOV,@V1,e1,e2 Move and animate as for WMOV, with detection, e1 times, with or without flyback synchronisation. Again, the detection option is unlikely to be of use.

JOYSTICK/KEYBOARD MOVE COMMANDS

There are four commands in this group and each allows a previously placed sprite to be moved from one screen position to another under keyboard or joystick control. As with the previously detailed MOVE commands, the only difference between them is the type of operation used to carry out the movement. The four commands in this category can be repeatedly executed and have a collision detection option. All joystick/keyboard MOVE commands are suffixed with MVJ.

Animation Sequences

The J/K (joystick/keyboard) MOVE commands provide exactly the same animation facilities as the previously described linear MOVE commands. If the joystick/keyboard is not activated then the movement routines are effectively handed zero values for the X and Y increment and no movement takes place. This means that the animation sequence does not increment either and this prevents the animation being out of step.

For each of the four commands:

Parameter	Use
SP1	Sprite to be removed.
SP2	Next sprite to be PUT.
SP3	Sprite to be PUT when SP2 is removed.
SP4	Sprite to be PUT when SP3 is removed.
COL	Screen column of sprite to be moved.

ROW	Screen row of sprite to be moved.
HGT	Y-increment of movement.
LEN	X-increment of movement.
@V1	Variable address for collision detection.
e1	Number of times to execute command.
e2	Frame-flyback synchronisation flag.
KEY	Physical row to poll for movement (see "GETTING STARTED").

Joystick/Keyboard Control

If the joystick/keyboard is not activated, no operation takes place.

If the joystick/keyboard is activated left, the sprite will move left by the value held in LEN. If LEN is negative it will actually move right. If LEN is zero it will not move or animate.

If the joystick/keyboard is animated right, the sprite will move right by the value held in LEN. If LEN is negative is will actually move left. If LEN is zero it will not move or animate.

If the joystick/keyboard is activated upward, the sprite will move upward by the value held in HGT. If HGT is negative it will actually move down. If HGT holds zero it will not move or animate.

If the joystick/keyboard is activated downward, the sprite will move downward by the value held in HGT. If HGT is negative it will actually move upward. If HGT holds zero it will not move or animate.

Combinations of the above will cause diagonal movement.

Command	Action
FMVJ	The same action as the linear move FMOV, i.e. animate in front of screen data, but the direction of movement is under joystick/keyboard control.
FMVJ,@V1	The same action as the linear move FMOV,@V1, i.e. animate in front of screen data, with collision detection, but the direction of movement is under joystick/keyboard control.
FMVJ,@V1,e1,e2	The same action as the linear move command FMOV,çV1,e1,e2 ,i.e. animate in front of screen data, e1 times, with or without flyback synchronisation. Direction of movement is under joystick/keyboard control.
BMVJ	The same action as the linear move command BMOV, i.e. animate behind screen data, but with the direction of movement under joystick/keyboard control.
BMVJ,@V1	The same action as the linear move command BMOV,@V1, i.e. animate behind screen data, with collision detection, but with the direction of movement under joystick/keyboard control.
BMVJ,@V1,e1,e2	The same action as the linear move command BMOV,çV1,e1,e2 ,i.e. animate behind screen data, e1 times, with or without flyback synchronisation. Direction of movement is under joystick/keyboard control.
XMVJ	The same action as the linear move command XMOV, i.e. animate using the exclusive-OR operation, but with the direction of movement under joystick/keyboard control.
XMVJ,@V1	The same action as the linear move command XMOV,@V1, i.e. animate using the exclusive-OR operation, with collision detection, but with the direction of movement under joystick/ keyboard control.
XMVJ,@V1,e1,e2	The same action as the linear move command XMOV,@V1,e1,e2 ,i.e. animate using the exclusive-OR operation, e1 times, with or without flyback synchronisation. Direction of movement is under joystick/keyboard control.
WMVJ	The same action as the linear move command WMOV, i.e. animate with the block over-write operation, but with the direction of movement under joystick/keyboard control.
WMVJ,@V1	The same action as the linear move command WMOV,@V1, i.e. animate using the block over-write operation, with collision detection, but with the direction of movement under joystick/ keyboard control.

BOUNCING MOVE COMMANDS

There are four commands in this group and each allows a previously placed sprite to be 'bounced' within the confines of a pre-defined rectangular window. The window is set using the BWST command which is described at the end of this section. Again, the difference between the four commands in the group is the type of movement employed. Each command in this category can be repeatedly executed and has a bounce detection option. All bouncing MOVE commands are suffixed with BNC.

Animation Sequences

The bouncing MOVE commands use SP1, SP2, SP3 and SP4 to provide an animation option. The 'frames' are rotated every time the command is executed unless the MOVE increments in HGT and LEN are both set to zero, in this case no action is taken. When a 'bounce' takes place, the animation sequence is automatically reversed. If bounce detection is 'on', then the user has the option to 're-reverse' the sequence using SWPS if a bounce is tested for and found to have occurred. If, however, the sprite bounces simultaneously on two edges, the result of the two 're-reversals' will cancel each other out and the animation sequence will not run in reverse. It should also be noted that a double bounce will still only increment the collision detection variable by 1 and there is no way of testing for a double bounce without examining the collision detection variable and SP2.

For each of the four commands:

Parameter	Use
KEY	Number of the sprite containing the bounce window parameters. Note that KEY and not SPN is used.
SP1	Sprite to be removed.
SP2	Next sprite to be PUT.
SP3	Sprite to be PUT when SP2 is removed.
SP4	Sprite to be PUT when SP3 is removed.
COL	Screen column of sprite to be removed.
ROW	Screen row of sprite to be removed.
HGT	Y-increment of movement.
LEN	X-increment of movement.
@V1	Variable address for collision detection.
e1	Number of times to execute command.
e2	Frame-flyback synchronisation flag.

Command	Action
FBNC	The same action as the linear move FMOV, i.e animate in front of screen data, but the direction of movement is subject to the bounce condition imposed by the pre-defined bounce window.
FBNC,@V1	The same action as the linear move FMOV,@V1, i.e. animate in front of screen data, this time with bounce detection, but direction of movement is subject to the bounce condition.
FBNC,@V1,e1,e2	The same action as the linear move FMOV,@V1,e1,e2, i.e. animate in front of screen data, with collision detection, e1 times, with or without flyback synchronisation. Direction of movement is subject to bounce condition.
BBNC	The same operation as FBNC, but movement is behind screen data.
BBNC,@V1	The same operation as FBNC,@V1, but movement is behind screen data.
BBNC,@V1,e1,e2	The same operation as FBNC,@V1,e1,e2 but movement is behind screen data.
XBNC	The same operation as FBNC, but movement is acheived by the XOR operation.
XBNC,@V1	The same operation as FBNC,çV1, but movement is acheived by the XOR operation.
XBNC,@V1,e1,e2	The same operation as FBNC,@V1,e1,e2, but movement is acheived by the XOR operation.
WBNC	The same operation as FBNC, but movement is acheived by the block over-write operation.

WBNC,@V1	The same operation as FBNC,çV1, but movement is acheived by the block over-write operation.
WBNC,@V1,e1,e2	The same operation as FBNC,@V1,e1,e2, but movement is acheived by the block over-write operation.

BWST

An additional command is required to set up the window in which 'bouncers' are constrained to move. The command is detailed in this section as it is only applicable to this group of operations.

Parameter	Use
KEY	The number of the sprite which is to contain the window information.
COL	The left hand column of the window. This is the furthest left the sprite will move.
ROW	The top row of the window. This is the lowest row value the sprite will take before bouncing.
LEN	This is the length of the window and is the horizontal distance the sprite will move, as measured from COL, before bouncing. Note that the sprite size will not affect the window size as it is the sprite co-ordinates which are constrained to move within the window.
HGT	This is the height of the window and is the vertical distance the sprite will move, as measured from ROW, before bouncing.

Command	Action
BWST	Set up bouncing window data in sprite KEY. Note that the sprite containing the data must have a height of 1 and a length of 4 or a parameter error will be issued.

THE DATA SCANNING COMMANDS

Three commands are provided for scanning screen windows, sprites and sprite windows for pixel data, and are complimentary to the collision detection facility. These can be used in conjunction with the logical GETs and PUTs to facilitate collision detection and pattern matching. If, for instance, a screen window is ANDed with a sprite, then the SCAN commands can be used to detect data which, if present, indicates a collision. If a screen window is XORed with a sprite then again, the SCAN commands can be used to detect data which, if absent indicates a pattern match. There are three data types (indicated by the command suffix) and these are:

Type V

The scan takes place over a screen window, where COL and ROW specify the screen position of the window top left, and HGT and LEN specify the window dimensions. The window parameters are adjusted if it lies partially "off-screen".

Type S

The scan takes place over the whole sprite whose number is held in SPN.

Type P

The scan takes place over a sprite window. The number of the sprite is given by SPN and the window is specified by COL, ROW, HGT and LEN. The window is adjusted if it lies partially "off-sprite".

Parameter	Use
COL	Screen column of window top left.
ROW	Screen row of window top left.
LEN	Window width.
HGT	Window height.
V1	Data detection variable.

Command	Action
SCNV,@V1	The screen window is scanned for pixel data, and if found, the integer variable V1 is incremented.
SUMV,@V1	The 16 bit sum of all the data in the screen window is formed and the result assigned to the BASIC variable V1.

SUMV,e1, e2,...@V1	Again the 16 bit sum of all the data in the screen window is formed. If more than one parameter is used however, the sum is compared with each expression in the list and the position in the list of the first expression to match the calculated sum is assigned to V1. If no match is found V1 is assigned 0. A maximum of 8 parameters (including çV1) is allowed.

Parameter	Use
SPN	The number of the sprite to be scanned.
V1	Data detection variable.

Command	Action
SCNS,@V1	The sprite is scanned for pixel data, and if found, the integer variable V1 is incremented.
SUMS,@V1	The 16 bit sum of all the data in the sprite is calculated and the result assigned to the BASIC variable V1.
SUMS,e1, e2,...@V1	This command works in exactly the same way as the previously described SUMV command except that the sprite whose number held in SPN is used instead of a screen window.

Parameter	Use
SPN	The number of the sprite containing the window to be scanned.
COL	The column of the window top left.
ROW	The row of the window top left.
LEN	Width of sprite window.
HGT	Height of sprite window.

Command	Action
SCNP,@V1	The sprite window is scanned for pixel data, and if found, the integer variable V1 is incremented.
SUMP,@V1	The 16 bit sum of all the data in the sprite window is formed and the result assigned to the BASIC variable V1.
SUMP,e1, e2,...@V1	This command works in exactly the same way as the previously described described SUMV and SUMS commands except that a sprite window is used as data for the sumation.

MISCELLANEOUS COMMANDS

FILL

This command is provided to fill irregular shapes with a particular INK. The user must specify the co-ordinates (in pixels) at which FILLing is to begin, and the number of the INK to FILL with. The area that contains the colour of the pixel at the chosen starting point will be filled with the indicated INK. In 4 colour mode the screen is 320 pixels wide, in 16 colour mode the screen is 160 pixels wide. In both cases the screen is 200 pixels high. The FILL command makes extensive use of the machine stack and if the object to be filled is particularly intricate the command may terminate before completing with ** OUT OF MEMORY **. For this reason use the command with great care. The command only works with one data type and this is the screen.

Parameter	Use
XCL	Pixel co-ordinate of starting point (0 to 319 in 4 colour mode or 0 to 159 in 16 colour mode) as measured from the left.
ROW	Pixel co-ordinate of starting point (0 to 199 in both colour modes) as measured from the top of the screen.
IK1	INK number to fill with (0 to 3 or 0 to 15).

Command	Action
FILL	The screen is filled from the starting point outward, bounded by pixels with a colour different from the starting point, or the screen edges.

SPNV

The screen window, indicated by COL, ROW, HGT and LEN is rotated by 90 degrees into a window whose dimensions are the transpose of the source window and whose top left is specified by SCL and SRW. The windows are adjusted to lie "on-screen". The source and target windows should not overlap or the results will be unpredictable, albeit spectacular!!! This command is provided for use with screen data only. Note that the height of the window to be 'SPUN' is rounded down to the nearest 8 pixels, i.e. HGTs of 25, 26, 27, 28, 29, 30 and 31 would all be rounded down to 24 - and so on.

Parameter	Use
COL	Screen column of source window top left.
ROW	Screen row of source window top left.
HGT	Height of window to be rotated. (Mod 8).
LEN	Width of window to be rotated.
SCL	Screen column of target top left.
SRW	Screen row of target top left.

Command	Action
SPNV	The screen window is rotated by 90 degrees in the clockwise direction, into a second screen window, with the transposed dimensions of the source window and the top left specified by SCL and SRW.

KBFN,@V1

This command scans the keyboard hardware (its BASIC equivalent scans the table which is updated on interrupt). Thus KBFN can be used to scan even if the interrupt is disabled. The command KBFN is followed by one parameter, which is the address of a BASIC integer variable V1. If the key whose number is specified in the graphics variable KEY is pressed, V1 is incremented, otherwise it is unchanged. Multiple key presses are thus enabled but if three keys on one row are pressed then the hardware may return a fourth as being pressed.

Parameter	Use
KEY	Number of key to be tested (see Appendix A).

Command	Action
KBFN,@V1	Test the key whose number is held in KEY and increment V1 if it is pressed.

SCLS

Clear the whole screen to INK 0 (paper colour), home the cursor to the top left, set the screen to window 0 with full width, and height 24 rows. The bottom row cannot be used by BASIC but all of the graphics routines will utilise the whole screen. If the user changes the current window to be the whole screen then hardware scrolling is enabled. If this happens then SCLS must be executed before any of the extended graphics commands. SCLS has no parameters.

Command	Action
SCLS	Clear the screen, home the cursor, make the screen the current stream and define a window with the width of the screen and height 24 rows (1 less than the full screen).

BILD

This command is used to set up screen backdrops and is really best explained by example. A data compression technique is used so that sprite building blocks are represented by bits in a data sprite. The command has 5 parameters. COL and ROW are used to specify where the building should begin from. Note that COL and ROW can be "off-screen" and the backdrop can be larger than the screen. In each case the command will fit as much data as it can onto the screen. The graphics variable TYP is used to hold the type of operation that should be used to PUT the data to the screen.

Value in KEY	Operation
0	Block overwrite
1	Exclusive-OR
2	PUT in front of current data
3	PUT behind current data

The graphics variable SP1 is used to hold the number of the sprite which is used as the building block and the variable SPN holds the number of the sprite which contains the BIT pattern that determines where the building blocks will be PUT.

The Data Sprite Format

The format for the data sprite (containing the bit pattern) is quite straightforward. The width in "sprites" of the scenario will be 8 times the width of the data sprite. This is because there are 8 bits per byte. The height of the scenario in "sprites" will be the height of the data sprite. The actual physical size of the scenario also depends on the size of the sprite which forms the building block. If, for instance, the sprite building block was 20 pixels high and 5 bytes wide and the data sprite was 2 bytes wide and 10 bytes high, then the backdrop would be 5x2x8=80 bytes wide and 20x10=200 pixels high and would exactly fill the screen. Thus if your basic building block is 20 pixels high and 5 bytes wide then you will need 20 bytes per "screen".

Parameter	Use
COL	Screen column to begin building.
ROW	Screen row to begin building.
KEY	Type of operation.
SPN	Number of data sprite (contains bit map).
SP1	Number of sprite to build with.

Command — Action

BILD — The data sprite, whose number is held in SPN, is scanned from left to right, a bit at a time. If a bit is set then the sprite whose number is held in SP1 is PUT to the screen and the column advanced by the width of the building sprite. If a bit is not set then the column is simply advanced by the width of the building sprite. The operation is repeated for each of the rows in the data sprite and each time the screen row is advanced by the height of the building sprite. The overall effect is just to expand from bit to sprite.

NOTE: — The gaps between building blocks (represented by bits set to 0) are not overwritten so data already present is not cleared. The data sprite can be scrolled using WSVN, WSL4, WSR4, WSL8, WSR8 without corrupting the data structure. Using the scrolls and suitable values for COL and ROW, users can create scenarios many screens high and wide and move a screen window through them. This is, however, an advanced technique and should not be attempted until familiarity is gained with the whole package.

DEEK and DOKE

DEEK and DOKE are the 16 bit equivalents of PEEK and POKE and are included in this package because of their omission from locomotive BASIC.

Command — Action

DEEK,e1,@V1 — 16 bit equivalent of PEEK. The contents of the address e1 are assigned to the BASIC integer variable V1.

Parameter	Use
@V1	Address of variable in the assignment.
e1	BASIC expression for the address e1.

Command — Action

DOKE,e1,e2 — 16 bit equivalent of POKE. The low byte of the value of e1 is placed in e1 and the high byte is placed in e1+1.

Parameter	Use
e1	Address to be 'DOKE'd.
e2	Data to 'DOKE'

TRACKING SPRITES

Although one of the main aims of the package throughout has been to keep the features as simple as possible, some complex features have been included for the more adventurous user. Before attempting to use tracking sprites, please ensure that you are fully conversant with the workings of the rest of the package. The nature of these commands means that they are not crash protected and the outcome of mistakes is unpredictable.

Format for Tracking Sprites

A tracking sprite, in the context of Laser Basic, is not really a sprite at all. A tracking sprite is really a primitive program which is held within a dummy sprite. The program controls where the sprite moves to, the type of operation used for its movement, and which sprites are actually used in the animation sequence. This provides much greater flexibility than the previously described MOVE commands and the advanced user will appreciate that almost entire programs could be written using tracking sprites alone. The format of the tracker is as follows:

BYTE USE

0,1 16 bit program counter. This is a pointer to the next instruction in the tracker sprite and is advanced by the length of the instruction each time an instruction is executed.

2 The screen column at which the sprite was last PUT.

3 The screen row at which the sprite was last PUT.

4 The number of the sprite that was last PUT.

5 The type of move operation that the tracker is currently using.
The type byte has four fields:

Bits 0 and 1 These specify the movement operation.
00 = Move in front of screen data.
01 = Move behind screen data.
10 = Move with the XOR operation.
11 = Move with block overwrite.
Bit 2 If bit 2 is set, then the increment will be joystick/keyboard controlled.
Bits 3,6 If bit 2 is set, then bits 3 to 6 specify the key row/joystick to be used.
Bit 7 If bit 7 is set, then collision/bounce detection is enabled.

After the initial 6 bytes of system information come the instructions themselves. If a sprite is simply to be moved and animated then the instructions will be 3 bytes long:

BYTE OPERATION

0 The number of the sprite to be PUT. The system knows the number of the last sprite PUT and if either XINC or YINC (see next paragraph) is non-zero then the last sprite to be PUT will be removed and the new sprite to be PUT will be given by this byte. If this byte is zero then the system does not execute a MOVE but instead looks for a control code (see Control Codes in next section).

1 This is the X-increment and is the amount by which to move in the horizontal direction. The MOVE command takes into account whether the system is in 80 or 160 column mode and positive values move right and negative values move left.

2 This is the Y-increment and is the amount by which to move in the vertical direction. Positive values move downward and negative values move upward.

NOTE: If the X-increment and the Y-increment are both zero the instruction is ignored and the program counter is advanced to point to the next instruction before returning.

Control Codes

If the control program finds that byte 0 (which it expects to be a sprite number) contains a zero, then it knows that a control code is to follow. There are four control codes in all, and these are:

Code Operation

0 Start at first instruction. The program counter is set to point to the first instruction in the tracker. This instruction is executed and the program counter is advanced to point to the second instruction before returning.

1 Start new track with sprite number equal to the value in the next byte of the current tracker. The last sprite PUT is removed using the current operation and then the new tracker sprite is launched. Tracking involves setting the program counter to point to the first instruction, setting the COL and ROW pointers to those of the sprite last removed and re-PUTting the sprite just removed but using the operation specified in the new tracker. For this reason, the new tracker must use the same type of sprite (masked or unmasked) as the old tracker. This code should only be used if the tracker is being executed from within a controller or the outcome is unpredictable.

Continue track with move type equal to the value in the next byte of the current tracker. The current sprite is removed from the screen using the current operation, and then put back using the new operation. Byte 5 of the tracker variables is amended to hold the new move type and then the next instruction in the current tracker is executed. Thus a code of 2 effectively executes 2 instructions and the program counter is therefore advanced by 6 bytes.

3 Start new track with sprite number equal to the value in the next byte of the current tracker and program counter equal to the value in the next two bytes after that. The effect is exactly the same as CODE 1 except that the new tracker can be entered at a convenient point and need not be run from the start. Again this code should only be used where the tracker is being executed from within a controller.

4 Execute the command whose address (see ADDR) is in the next 2 bytes but one, using the variable set whose number is in the next byte. For details of command addresses see the section on compiler related commands.

Launching a Tracker

Before a tracker can be executed using the TMOV command, it needs to be "launched" using the TPUT command. The TPUT command requires 5 parameters. The column and row at which to start the track, the number of the first sprite to PUT, the number of the tracker and the value for the program counter at which execution should begin. The first sprite is PUT to the screen and the system information set up in the required tracker.

Parameter	Use
COL	Screen column of start of track.
ROW	Screen row of start of track.
KEY	Holds the number of the sprite holding the tracker program.
SP1	First sprite to PUT.
e1	Initial program counter value (1 = first instruction in tracker).

Command	Action
TPUT,e1	Tracker KEY is initialised and sprite SP1 is placed at the screen position specified by COL,ROW using the operation specified by the tracker sprite KEY. The program counter in the tracker KEY is set to the value of the expression e1. The tracker can now be executed using the TMOV command.

Moving a Tracker

Parameter	Use
KEY	The number of the tracker, one instruction of which is to be executed.
@V1	Optional parameter which is the address of the BASIC integer variable which is to be incremented if collision detection is enabled and detected.

Command	Action
TMOV	Execute one instruction of the tracker whose number is held in KEY.
TMOV,@V1	Execute one instruction of the tracker whose number is held in KEY. If collision detection is enabled (BIT 7 of move type) and if a collision is detected, increment the BASIC integer variable V1.

CONTROLLERS

There are two controller commands, CPUT and CMOV. These are analogous to the TPUT and TMOV commands but work with sets of trackers instead of individual trackers. In each case the information is held in sprites so the only parameter required is the number of the sprite containing the controller information.

CPUT

The CPUT command launches all the trackers in the list contained in the data sprite whose number is held in SPN. The format for the controller holding the launch information is as follows:

BYTE	Use
0	Number of tracker to be launched.
1	Screen column to launch at.
2	Screen row to launch at.
3	Number of first sprite to PUT.
4,5	16 bit offset into tracker, to begin execution at.

Bytes 6 to 11 contain the information for the second tracker to launch, 12 to 17 the next, and so on. The end of the list is marked with a zero byte so that the number of the sprite to be launched is read as zero.

Parameter	Use
SPN	Number of the sprite containing the launch information.

Command	Action
CPUT	The set of trackers whose launch information is contained in the sprite whose number is held in SPN, are launched. This means effectively a series of TPUTs are executed using the parameters contained in the controller.

CMOV

Once a set of trackers has been launched using the CPUT command, the set can be executed using the CMOV command. The format for the controller is as follows:

BYTE	Use
0	Number of the tracker to execute.
1	This byte is set to 2 if collision detection is 'on' in the tracker, and if a collision occurred. If a collision does not occur on the next CMOV it is re-set to 0.

Bytes 2 to 3 hold the second tracker to be executed and so on. Again the list is terminated with a zero.

Parameter	Use
SPN	The number of the sprite containing the list of trackers to execute.

Command	Action
CMOV	Execute an instruction in each of the trackers contained in the data sprite whose number is held in SPN.
NOTE:	If a tracker is executed with the instruction "start new tracker" it will automatically modify the data sprite to contain the number of the new tracker.

BACKGROUND EXECUTION

One of Laser BASIC's most unique and powerful features is its ability to execute repeatable commands under interrupt. This means that a list of commands, each with its own associated variable set, can be run in background (synchronised to the frame-flyback) while the normal BASIC program runs in foreground. Using this facility will provide the user with the optimum speed and smoothness.

Command — Action

**ISET,@V$** — The commands and their variable sets contained in the string variable V$ are compiled into the background table. The format for the string is as follows:

Bytes	Use
0,3	First command.
4	Variable set for first command.
5,8	Second command.
9	Variable set for second command.
▪	
▪	
▪	
N,N+3	Last command.
N+4	Variable set for last command.

NOTE: The required variable set is selected by using one of the 16 letters A to P. The last character in the string must be a "#".

Example: "WVR1AINVVBXMOVC#" would compile into the table the commands WVR1, INVV and XMOV. WVR1 would use SET 0, INV would use SET 1 and XMOV would use SET 2. The commands would execute in the above order and the execution of the foreground program would be completely halted until the completion of the full background task.

Command — Action

IRUN,e1 — Begins execution of the background program which was compiled using ISET. The value of the BASIC expression e1 is used to monitor the frequency of the background program execution. If e1 is zero then the program will execute on every flyback (50 times a second), if e1 is 1 then it will allow one flyback between executions and so on up to 65535 which will only execute every twenty minutes or so.

NOTE: IRUN should never be executed unless ISET has been executed previously. IRUN should never be executed if a background program is already running. To ensure that no background program is executing use IEND (see next section) to halt the background task, or press the "ESC" key. If the execution time for the task is longer then 1/50th of a second and e1 is 0, then control will never return from the background program. This may be the desired effect, but pressing the "ESC" key will always halt the execution of the background task and thus prevent the system from becoming 'locked-up'. Errors in the extended commands running in foreground will also halt the background task.

Command — Action

IEND — Halts execution of the background task. This command should always be executed before re-executing ISET or IRUN.

For a list of commands which can be executed in background refer to the command summary.

COMPILER RELATED COMMANDS

With the exception of ADDR which is used with tracking sprites, these commands will probably only be used if you intend to compile the Laser BASIC program you are writing, or are short of space.

The requirement for these extra commands stems from the fact that the RSX command table is not part of the final compiled run-time program.

ADDR,@V1,@A$

This command assigns to the BASIC integer variable V1, the execution address of the command whose character string is in A$. For example to find out the execution address of WVR1 use:

```
        A$="WVR1":|ADDR,@X%,@A$:PRINT X%
```

The command must be entered in upper case characters or "Illegal Command" will be reported. The address returned is generally used as data for tracking sprites when control code 4 is employed. ADDR cannot be compiled by the Laser BASIC compiler.

IPUT and IGET

Due to the fact that the RSX table is not present in compiled programs, ISET cannot be compiled either. This would prevent background programs from being executed and so 2 additional commands have been included.

IPUT

IPUT will store the current interrupt table (created by ISET) into the sprite whose number is held in SPN.

IGET

IGET will retrieve the interrupt table data from the sprite whose number is held in SPN.

If you intend to compile your Laser BASIC program then all you need to do is execute the ISET command to set up the interrupt table and then execute IPUT to save the information into a sprite. The sprite can be loaded into the compiled program along with the other sprites and then IGET used to restore the table to the state it was formerly in. Out of memory errors will occur if the sprite is not large enough to accomodate the table (three bytes per command in the interrupt list plus one delimiter byte) or if a table is loaded which overflows the interrupt table space (90 bytes).

LASER BASIC SOUND

This final section deals with what is probably the most difficult to use feature of Laser BASIC. The Amstrad's own BASIC caters for sound control in a much more 'simple to use' manner and this extended facility need only be used in very advanced applications. As with tracking sprites, the nature of the operation means that there is very little crash protection and the outcome of mistakes is unpredictable. These commands do not have to be driven under interrupt, but in practice this will usually be the case.

The philosophy employed in Laser BASIC's sound handling is very similar to that used by tracking sprites - the data sprite contains a primitive program, in this case there are 20 instructions in the set. The program in the data sprite does not have to be run under interrupt but for most applications it almost certainly will be. To run sound under tracking sprite control the PLAY command is entered as a command in a normal tracker sprite (see control code 4 in the section on tracking sprites).

Command	Action
PLAY,e1,e2	The 16 bit program counter in the sound program contained in the sprite whose number is held in KEY is set to e1. The execution frequency is set to e2 but the sound program itself is not executed.
PLAY	The sound program contained in the sprite whose number is held in KEY is executed.

Parameter	Use
KEY	Holds the number of the sprite containing the sound program.

Format for the Program

BYTE	USE
0,1	16 bit program counter. This is a pointer to the next instruction in the data sprite and is advanced by the length of the instruction each time an instruction is executed. Unlike tracker sprites, sound programs will execute until they reach an instruction which halts execution.

2 This is an 8 bit counter which is incremented every time PLAY is executed. The sound program will only execute when this value is equal to the 'limit' held in BYTE 3. This facility is provided because sound programs usually run much more slowly than tracking sprite programs and means that they can be executed on selected tracker executions.

3 This is the 8 bit 'limit' which sets the execution rate of the sound program. Selecting a limit of 0 will cause execution to begin every time PLAY is executed, selecting a limit of 1 will cause execution to begin after every other PLAY and so on.

After these initial four bytes of information come the program itself. The first instruction (byte 4) corresponds to a program counter value of 1. Let's now look at the instructions themselves.

CODE	INSTRUCTION	LENGTH OF DATA (INCLUDING CODE)
0	SOUND	10
1	WAIT-SOUND	10
2	RESET	1
3	RELEASE	2
4	HOLD	1
5	CONTINUE	1
6	AMP-ENV	VARIABLE — UP TO 18
7	TONE-ENV	VARIABLE — UP TO 19
8	RE-RUN	1
9	JUMP	3
10	RE-LIM	3
11	CALL-CHANNEL	2
12	CALL-AMP-ENV	2
13	CALL-TONE-ENV	2
14	CALL-TONE-PERIOD	3
15	CALL-NOISE-PERIOD	2
16	CALL-INITIAL-AMP	2
17	CALL-DURATION	3
18	CALL-TONE-DURATION	5
19	STOP	1

To fully appreciate the operation of each of these instructions you should refer to the Amsoft firmware or BASIC manuals, but a sufficient description of each command is given in the next section. Where relevant, the entry point into the firmware JUMP table is given. .

SOUND - JUMP table #BCAA

This instruction attempts to add a 9 byte 'sound' to the sound queue of one or more channels. If any of the queues is full then no action will take place and the sound program will continue at the next instruction. The format for the 9 bytes of the sound are as follows:

BYTE	USE	POSITION IN LIST OF BASIC "SOUND" COMMAND
0	Channel status	1
1	Amplitude envelope	5
2	Tone envelope	6
3,4	Tone period	2
5	Noise period	7
6	Volume	4
7,8	Duration	3

For a full description of the range of each of the above parameters and their use in defining the sound produced, consult your BASIC manual or the "SOUND" section of this manual.

WAIT-SOUND

This instruction works in the same way and uses the same parameters as the sound instructions but differs in one respect. If any of the queues that the instruction is trying to use are full then the execution of the PLAY command is terminated and the program counter is left pointing at the current 'WAIT-SOUND' instruction. In fact, 'WAIT-SOUND' is almost invariably used in place of 'SOUND'.

RESET - JUMP table #BCA7

This instruction stops the current sound and clears all sound queues.

RELEASE - JUMP table #BCB3

This instruction will RELEASE individually held sounds. It uses one byte of data which specifies which channels to release. Only 3 bits of the byte are used - see the BASIC command RELEASE.

HOLD - JUMP table #BCB6

This instruction freezes all sounds. These will be automatically re-started by the execution of SOUND, WAIT-SOUND, RELEASE or CONTINUE.

CONTINUE - JUMP table #BCB9

Restarts all sounds which have been held.

AMP-ENV - JUMP table #BCBC

This is analogous to BASIC's ENV command. It uses up to 18 bytes of data and the format for the data block is as follows:

BYTE	USE
0	Envelope number
1	Number of sections
2,3,4	Step count, step size, pause time for section 1
5,6,7	Step count, step size, pause time for section 2
•	
•	
•	
14,15,16	Step count, step size, pause time for section 5

Note that data is only supplied for the number of sections indicated in byte 1, and this is therefore a variable length instruction. For a full description of the parameters and their use, see the description of BASIC's ENV command.

TONE-ENV - JUMP table #BCBF

This is analogous to BASIC's ENT command. It uses up to 18 bytes of data and the format for the data block is as follows:

BYTE	USE
0	Envelope number
1	Number of sections
2,3,4	Step count, step size, pause time for section 1
5,6,7	Step count, step size, pause time for section 2
•	
•	
•	
14,15,16	Step count, step size, pause time for section 5

Again data is variable length and must correspond to the number of sections dictated by byte 1. For a comprehensive description see BASIC's ENT command.

RE-RUN

This instruction causes control to jump to the first instruction in the program, and execution to continue from that point. This provision allows sound programs to be indefinitely repeated.

JUMP

This 3 byte instruction is analogous to BASIC's GOTO or the Z80 s JP instructions. The two bytes forming the 16 bit program counter value to jump to are in LSB, MSB order as are all the 16 bit values used in this section.

RE-LIM

This instruction is provided to control the overall speed and phase of the program. It uses two data bytes which are passed to bytes 2 and 3 of the sound program. This has the effect of re-setting the count and limit. By passing a smaller limit the program can be made to execute with a higher frequency (although this doesn't usually cause the tune to play any faster!) and vice-versa. Passing a count equal to the limit will ensure that the program executes on the next invocation.

CALL Commands

There are actually 7 CALL instructions in all and in order to use these you must ensure that the first instruction in the program is a SOUND (code 0) or a WAIT-SOUND (code 1), otherwise an "ILLEGAL TRACKER CODE" error will be generated. As has been discussed, a 'SOUND' is represented by 9 bytes. What these instructions do is to modify 1 or more bytes in the 'SOUND', and then return to the point in the program from whence it was called. If CALL-TONE-PERIOD or CALL-TONE-DURATION is executed then the SOUND or WIAT-SOUND will be automatically executed. If the instruction being called is a SOUND instruction, then control will continue at the next instruction after the CALL whether the 'SOUND' was added to the queue or not. If the instruction was a WAIT-SOUND then control will only continue at the next instruction after the CALL if the 'SOUND' was successfully added. If it was not, then the program counter remains pointing at the CALL and the program is terminated. If the latter was the case then the next time PLAY is executed with count equal to the limit then this CALL will be the first instruction to execute.

CALL-CHANNEL

Calls the first instruction in the program which must be a SOUND or WAIT-SOUND. The channel status in the 9 byte 'SOUND' block is replaced by the new channel status and the sound executed. The new channel status is left in the data block after control returns but a SOUND or WAIT-SOUND is not executed.

CALL-AMP-ENV

This works in the same way as CALL-CHANNEL except that the new data is a volume envelope number.

CALL-TONE-ENV

This works in the same way as CALL-AMP-ENV except that the data is a tone envelope number.

CALL-TONE-PERIOD

This works in the same way as CALL-TONE-ENV except this time the data is 2 bytes long and represents the tone period. The other difference is that this instruction will cause the SOUND or WAIT-SOUND to execute with the modified TONE-PERIOD.

CALL-NOISE-PERIOD

This works in the same way as CALL-TONE-ENV except that the data is 1 byte long again and represents the Noise period.

CALL-INITIAL-AMP

Works in the same way as CALL-NOISE-PERIOD except that the data represents the starting volume (which can be altered by the volume envelope if one is specified).

CALL-DURATION

Works in the same way as the other commands but its 2 data bytes specify the new duration.

CALL-TONE-DURATION

Works in the same way as CALL-TONE-PERIOD but this instruction has 4 data bytes, the first two represent a new duration and the third and fourth represent a new tone period. As with CALL-TONE-PERIOD, the SOUND or WAIT-SOUND instruction being called is executed.

NOTE: All two byte numbers are entered in least significant, most significant byte order.

STOP

This will cause the termination of the sound program and a return to be made to wherever 'PLAY' was executed from (usually a tracker). The program is exited with the program counter pointing to the instruction after STOP, which will be the next instruction to be executed the next time PLAY is executed.

This concludes the final section of the Laser BASIC commands in detail, for more examples of interrupt driven sound see the section on SOUND in the main part of this manual.

LASER BASIC ERRORS

Error 1 — ** SPN TOO HIGH **

This error occurs whenever a sprite number is being used with a value greater than SMAX. The variables which hold the sprite numbers are SPN, SP1, SP2, SP3, SP4 and KEY. Sprite numbers are also held in tracking sprites. Be sure that you are using the correct SET particularly when using background programs, tracking sprites, EXXV, ASTV, AVTS and ESAV.

Error 2 — ** SPN EXISTS **

This occurs when an attempt is made to add a sprite to the table with a number that has been previously allocated. It can occur with CSPR, HRSP, RNUM, ADNM and MSPR. This error is usually straightforward to identify.

Error 3 — ** SPN OF ZERO **

This error occurs when an attempt is made to use sprite number zero and can occur in numerous instances. The best way to deal with this error is to look at the values held in SPN, SP1, SP2, SP3, SP4 and KEY. Although this error has many possible causes it is usually fairly obvious where to look.

Error 4 — ** SMAX OF ZERO **

This occurs if an SMAX of zero is given as the first parameter of SSPR. This error seldom occurs as SSPR is seldom used.

Error 5 — ** ZERO LENGTH DATA **

This error very rarely occurs and signals an attempt to block move a zero length of code. It can happen if an attempt is made to relocate an empty sprite file, or if IPUT/IGET are used to move an empty interrupt list, but generally indicates that the sprite table or system variables have become corrupted. It is unwise to continue after the error unless you are sure of it's cause.

Error 6 — ** SPN DOESN'T EXIST **

Again this error can result from numerous different situations. By examining the variables SPN, SP1, SP2, SP3, SP4 and KEY it is usually possible to work out where the error occured.

Error 7 — ** CAN'T MASK ****

An attempt has been made to MASK, re-MASK or de-MASK a sprite with an 'odd' as opposed to 'even' width (in bytes).

Error 8 — ** ILLEGAL TRACKER TERMINATOR **

An illegal control code has been found in a tracker or sound program.

Error 9 — ** OUT OF MEMORY **

An attempt has been made to carry out an operation which requires more space than is available between MBOT (which should be HIMEM+1) and the bottom of sprites or has run out of stack space. To overcome this you will need to reduce HIMEM with the MEMORY command (followed by an MSET command), delete one or more of the existing sprites or relocate spites upwards (if this is possible). This error can also be caused by FILL if insufficient stack space is available but there is no way around this as the space allocated to a the stack is fixed. The following commands require free memory:

WVVN, WSVN, WPVN, HRSP, ISET, GSPR and MSPR

IPUT and IGET may also generate this error see the section on "compiler related commands".

Note: The BASIC 'LOAD' command uses a buffer which is not required after LOADing. This means that you can usually reduce HIMEM (don't forget the MSET!), once your BASIC program has been loaded. Sprites can then be loaded into a larger space.

Error 10 — ** ILLEGAL FILENAME **

This error occurs when an attempt is made to load, save or merge sprite files using a filename that does not end in "SPR" or "SPR.BAK". Filenames must be typed in upper case. If an error occurs during one of these 3 operations you may find that the filename has been altered. If this occurs you will need to re-type the filename.

Error 11 - ** PARAMETER ERROR **

This error occurs if a Laser BASIC command is followed by the wrong number of parameters or one of the SUM commands is followed by more than 8 parameters.

Error 12 — ** NO SPRITES **

This error occurs if an attempt is made to save an empty file of sprites.

Error 13 — ** OUT OF RANGE **

This error will occur if an attempt is made to assign one of the Laser BASIC variables with an illegal value. This error will also occur if an attempt is made to set up a bounce window in a sprite which does not have a height of 1 and a width of 4. Similarly this error will occur if ASTV, AVTS or ESAV are used with sprites that do not have a height of 1 and a width of 20.

Error 14 — ** ILLEGAL COMMAND **

This error can occur when using the ISET command and means that one of the commands in the string doesn't exist, or has be typed in lower case, or '#' is missing from the end of the string or a SET has been selected without using one of the letters 'A' to 'P' (in upper case). It can also occur when using the ADDR command.

COMMAND SUMMARY

The following is a summary of Laser BASIC's extended commands. The number after the command in the table is called the command class. Below is a description of each class.

CLASS 1

Commands in this class can be used in MODE 2 (2 colour mode) as well as MODEs zero and one. These commands do not have a collision detection option, cannot be included in tracker sprites, cannot be executed in background (see ISET) and do not have a repeat option. The following commands are included in Class 1:

```
ADDR    COL     COLQ    DEEK    DOKE    HGT     HGTQ    IK1     IK1Q    ISET
ISPR    KBFN    KEY     KEYQ    LEN     LENQ    MSET    MSPR    NPX     NPXQ
PSPR    ROW     ROWQ    RSPR    SCL     SCNP    SCNS    SCNV    SET     SETQ
SP1     SP1Q    SP2     SP2Q    SP3     SP3Q    SP4     SP4Q    SPN     SPNQ
SRW     SRWQ    SSPR    SUMP    SUMS    SUMV    TPUT    XCL     XCLQ
```

CLASS 2

Commands in this class cannot be used in MODE two and can only be used in MODEs zero and one. They do not have a collision detection option but could be used as instructions in tracker sprites. They cannot be executed in background and do not have a repeat option.

```
FILL    HRSP    MASK    MGXP    MGXS    MGXV    ONHI    ONLO    RMSK    SPNV
```

CLASS 3

Commands in this class can be used in MODE two as well as MODEs zero and one, do not have a collision detection option, can be used as instructions in a tracker, cannot be executed in background and do not have a repeat option.

```
ADNM    CSPR    DMSK    DSPR    ESPR    GSPR    IEND    MGYS    MGYV    RNUM
```

CLASS 4

Commands in this class can be used in all three graphics modes, do not have a collision detection option, can be executed in trackers, can be executed in background but do not have a repeat option.

```
ASTV    AVTS    BILD    BWST    CLHI    CLLO    CLSP    CLSS    CLSV    CPUT
ESAV    EXXV    FIPS    FIPV    FOPS    FOPV    IGET    INVP    INVS    INVV
IPUT    PLAY    SCLS    STCP    STCS    STCV    SWPS
```

CLASS 5

Commands in this class do not work in MODE two, do not have a collision detection option, can be executed in trackers, can be executed in background but do not have a repeat option.

```
MIRP    MIRS    MIRV    MORP    MORS    MORV    SETP    SETS    SETV
```

CLASS 6

Commands in this class do not work in MODE two, but do have a detection option, can be executed in trackers, can be executed in background, but do not have a repeat option.

```
BGET    BPUT    FSWP    GMBH    GMIF    GTBH    GTIF    GWBH    GWIF    PMBH
PMIF    PTBH    PTIF    PWBH    PWIF
```

CLASS 7

Commands in this class work in all three graphics MODEs, do have a detection option, can be executed in background, but do not have a repeat option.

```
GMBL    GMND    GMOR    GMXR    GTBL    GTND    GTOR    GTXR    GWBL    GWND
GWOR    GWXR    PMBL    PMND    PMOR    PMXR    PTBL    PTND    PTOR    PTXR
PWBL    PWND    PWOR    PWXR
```

CLASS 8

Commands in this class do not work in graphics MODE two, do not have a detection option, but can be executed in trackers, can be executed in background and do have a repeat option.

```
SPL1   SPR1   SSL1   SSR1   SVL1   SVR1   WPL1   WPR1   WSL1   WSR1
WVL1   WVR1
```

CLASS 9

Commands in this class do work in graphics MODE two, do not have a detection option but can be executed in trackers, are background executable and do have a repeat option.

```
CMOV   SPL4   SPL8   SPR4   SPR8   SPVN   SSL4   SSL8   SSR4   SSR8
SSVN   SVL4   SVL8   SVR4   SVR8   SVVN   WPL4   WPL8   WPR4   WPR8
WPVN   WSL4   WSL8   WSR4   WSR8   WSVN   WVL4   WVL8   WVR4   WVR8
WVVN
```

CLASS 10

Commands in this class do not work in MODE two but do have a detection option, can be executed in a tracker, can be executed in background and do have a looping option.

```
BBNC   BMOV   BMVJ   FBNC   FMOV   FMVJ
```

CLASS 11

Commands in this class do work in MODE two, do have a detection option, can be executed in a tracker, can be executed in background and have a repeat option.

```
TMOV   WBNC   WMOV   WMVJ   XBNC   XMOV   XMVJ
```

CLASS OPTIONS

The following is a summary of available options and a summary of the classes supporting them.

MODE 2

All Laser BASIC commands will execute in MODE zero (16 colour mode) and MODE one (4 colour mode), but only selected commands will function correctly with MODE 2 (2 colour mode). The following classes will work in MODE 2:

Class 1, class 3, class 4, class 7, class 9, class 11.

Collision Detection Option

Laser BASIC commands which move data around the screen and between sprites have a collision detection option. This option will never work on MODE two data. The following classes support the option:

Class 6, class 7, class 10, class 11.

Tracking Option

Laser BASIC commands can be executed from within tracking sprites (see Tracking Sprites, Control Code 4). Only certain commands can be executed in this way and only commands in the following classes should be used:

Class 2, class 3, class 4, class 5, class 6, class 7, class 8, class 9, class 10, class 11.

Background Execution

Certain commands can be compiled into an interrupt table and executed in background under interrupt. The following command classes support this option:

Class 4, class 5, class 6, class 7, class 8, class 9, class 10, class 11.

Repeat Option

Certain commands can be repeatedly executed in a machine code loop with or without frame-flyback synchronisation. The following classes support this option.

Class 8, class 9, class 10, class 11.

COMMAND/CLASS		PARAMETERS	ACTION
ADDR	1	@V1,@V$	Assigns the execution address of the command in the 4 character string V$ to the BASIC integer variable V1.
ADNM	3	SPN	Increment all existing sprite numbers by the value held in SPN.
ASTV	4	SET,SPN	The 20 bytes in sprite SPN are assigned to the current variable set.
AVTS	4	SET,SPN	The current variable set is assigned to sprite SPN.
BBNC	10	SP1,SP2,SP3	Bounce 'behind' screen data. SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2
BGET	6	SPN,COL,ROW @V1	Remove a previously 'BPUT' sprite.
BILD	4	COL,ROW,SPN SP1,KEY	Expand BIT pattern.
BMOV	10	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,@V1 e1,e2	Move linearly 'behind' screen data.
BMVJ	10	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2	Move under keyboard/joystick control, 'behind' screen data.
BPUT	6	SPN,COL,ROW @V1	'PUT' a sprite 'behind' screen data.
BWST	4	KEY,COL,ROW HGT,LEN	Bounce window data is set up in sprite KEY.
CLHI	4		Puts the software into 160 column mode.
CLLO	4		Puts the software into 80 column mode.
CLSP	4	SPN,COL,ROW HGT,LEN	Clear sprite window to INK 0.
CLSS	4	SPN	Clear whole sprite to INK 0.
CLSV	4	COL,ROW,HGT LEN	Clear screen window to INK 0.
CMOV	9	SPN	Execute one instruction in each of the trackers listed in SPN.
COL	1	e1	Assign the value of the expression e1 to the graphics variable COL.
COLQ	1	@V1	Assign the value in the graphics variable COL to the BASIC integer variable V1.
CPUT	4	SPN	Launch all of the trackers listed in SPN.
CSPR	3	SPN,HGT,LEN	Create sprite SPN with height HGT and length LEN.
DEEK	1	e1,@V1	The 16 bit contents of address e1,e1+1 are assigned to the BASIC integer variable V1.
DMSK	3	SPN	Sprite SPN is de-masked.
DOKE	1	e1,e2	The 16 bit value e2 is POKEd into e1 and e1+1 in LSB,MSB order.
DSPR	3	SPN	Sprite SPN is deleted.

ESAV	4	SET,SPN	Exchange the current SET with the contents of sprite SPN.
ESPR	3	SPN	Expand/contract sprite table space.
EXXV	4		Exchange current and alternative SET pointers.
FBNC	10	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2	Bounce 'in-front of' screen data.
FILL	2	XCL,ROW,IK1	Fill the shape with INK IK1.
FIPS	4	SPN	Vertically reflect sprite SPN.
FIPV	4	COL,ROW,HGT LEN	Vertically reflect screen window.
FMOV	10	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,@V1 e1,e2	Move linearly 'in-front of' screen data.
FMVJ	10	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2	Move under keyboard/joystick control 'in-front of' screen data.
FOPS	4	SPN	Make sprite SPN vertically symmetric.
FOPV	4	COL,ROW,HGT	Make screen window vertically symmetric. LEN
FREE	1	@V1	Calculate free space and assign result to the BASIC integer variable V1.
FSWP	6	COL,ROW,SPN @V1	Exchange sprite and screen data in front of screen data.
GMBH	6	SP1,SP2,SRW SCL,@V1	Place sprite SP1 'behind' window SP2.
GMBL	7	SP1,SP2,SRW SCL,@V1	Block move sprite SP1 into window in SP2.
GMIF	6	SP1,SP2,SRW SCL,@V1	Place sprite SP1 'in-front of' window in SP2. data in SP1.
GMND	7	SP1,SP2,SRW SCL,@V1	AND sprite SP1 into window in SP2.
GMOR	7	SP1,SP2,SRW SCL,@V1	OR sprite SP1 into window in SP2.
GMXR	7	SP1,SP2,SRW SCL,@V1	XOR sprite SP1 into window in SP2.
GSPR	3	@V$	Load sprites from DISK/TAPE.
GTBH	6	SPN,COL,ROW @V1	'GET' screen data 'behind' sprite data.
GTBL	7	SPN,COL,ROW @V1	Block move screen window into sprite.
GTIF	6	SPN,COL,ROW @V1	'GET' screen data 'in-front of' sprite data.
GTND	7	SPN,COL,ROW @V1	AND screen data into sprite.
GTOR	7	SPN,COL,ROW @V1	OR screen data into sprite.
GTXR	7	SPN,COL,ROW @V1	XOR screen data into sprite.

GWBH	6	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	'GET' screen window 'behind' sprite window.
GWBL	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	Block move screen window into sprite window.
GWIF	6	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	'GET' screen window 'in-front of' sprite window.
GWND	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	AND screen window into sprite window.
GWOR	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	OR screen window into sprite window.
GWXR	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	XOR screen window into sprite window.
HGT	1	e1	Assign the value of e1 to the graphics variable HGT.
HGTQ	1	@V1	Assign the value in the graphics variable HGT to the BASIC integer variable V1.
HRSP	2	SPN	Create hi-res pair in SPN and SPN+1.
IEND	3		Terminate execution of the background program.
IGET	4	SPN	Move data from sprite SPN into the interrupt table.
IK1	1	e1	Assign the value of e1 to the graphics variable IK1.
IK1Q	1	@V1	Assign the value in the graphics variable IK1 to the BASIC integer variable V1.
IK2	1	e1	Assign the value of e1 to the graphics variable IK2.
IK2Q	1	@V1	Assign the value in the graphics variable IK2 to the BASIC integer variable V1.
INVP	4	SPN,COL,ROW HGT,LEN	Invert (1's complement) sprite window.
INVS	4	SPN	Invert (1's complement) whole sprite.
INVV	4	COL,ROW,HGT LEN	Invert (1's comlement) screen window.
IPUT	4	SPN	Move the interrupt table into sprite SPN.
IRUN	1	e1	Set background program running with execution interval e1.
ISET	1	@dV$	Compile all the commands/sets in the string V$ into the interrupt table.
ISPR	1	SPN,HGT,LEN @V1,@V2,@V3 @V4	Interrogate sprite SPN.
KBFN	1	KEY,@V1	Increment the BASIC integer variable V1 if KEY is pressed.
KEY	1	e1	Assign the value of e1 to the graphics variable KEY.
KEYQ	1	@V1	Assign the value in the graphics variable KEY to the BASIC integer variable V1.

LEN	1	e1	Assign the value e1 to the graphics variable LEN.
LENQ	1	@V1	Assign the value in the graphics variable LEN to the BASIC integer variable V1.
MASK	2	SPN	Convert sprite SPN to become a MASKed sprite.
MGXP	2	SPN,COL,ROW HGT,LEN	X-expand sprite window.
MGXS	2	SPN	X-expand whole sprite.
MGXV	2	COL,ROW,HGT LEN	X-expand screen window.
MGYS	3	SPN	Y-expand whole sprite.
MGYV	3	COL,ROW,HGT LEN	Y-expand screen window.
MIRP	5	SPN,COL,ROW HGT,LEN	Horizontally mirror sprite window.
MIRS	5	SPN	Horizontally mirror whole sprite.
MIRV	5	COL,ROW,HGT LEN	Horizontally mirror screen window.
MORP	5	SPN,COL,ROW HGT,LEN	Make sprite window horizontally symmetric.
MORS	5	SPN	Make whole sprite horizontally symmetric.
MORV	5	COL,ROW,HGT LEN	Make screen window horizontally symmetric.
MSET	1	e1	Set lowest address usable by Laser BASIC to be e1.
MSPR	1	@V$	Merge sprite files.
NPX	1	e1	Assign the value of e1 to the graphics variable NPX.
NPXQ	1	@V1	Assign the value in the graphics variable NPX to the BASIC integer variable V1.
ONHI	2		Put Laser BASIC into 4 colour mode.
ONLO	2		Put Laser BASIC into 16 colour mode.
PLAY	4	KEY,e1,e2	Execute sound program.
PMBH	6	SP1,SP2,SCL SRW,@V1	GET data in window SP2 behind data in SP1.
PMBL	7	SP1,SP2,SCL SRW,@V1	Block move window in SP2 into sprite SP1.
PMIF	6	SP1,SP2,SCL SRW,@V1M	PUT data in window SP2 'in-front of' data in SP1.
PMND	7	SP1,SP2,SCL SRW,@V1	AND window in SP2 into sprite SP1.
PMOR	7	SP1,SP2,SCL SRW,@V1	OR window in SP2 into sprite SP1.
PMXR	7	SP1,SP2,SCL SRW,@V1	XOR window in SP2 into sprite SP1.
PSPR	1	@V1	'PUT' sprite file to TAPE/DISK.
PTBH	6	SPN,COL,ROW @V1	'PUT' sprite 'behind' screen data.
PTBL	7	SPN,COL,ROW @V1	Block move sprite into screen.

PTIF	6	SPN,COL,ROW @V1	'PUT' sprite 'in-front of' screen data.
PTND	7	SPN,COL,ROW @V1	AND sprite into screen data.
PTOR	7	SPN,COL,ROW @V1	OR sprite into screen data.
PTXR	7	SPN,COL,ROW @V1	XOR sprite into screen data.
PWBH	6	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	'PUT' sprite window 'behind' screen window.
PWBL	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	Block move sprite window into screen window.
PWIF	6	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	'PUT' sprite window 'in-front of' screen window.
PWND	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	AND sprite window into screen window.
PWOR	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	OR sprite window into screen window.
PWXR	7	SPN,COL,ROW SCL,SRW,HGT LEN,@V1	XOR sprite window into screen window.
RMSK	2	SPN	Re-MASK a previously MASKed sprite.
RNUM	3	SP1,SP2	Re-number sprite SP1 to become sprite SP2.
ROW	1	e1 ROW.	Assign the value of e1 to the graphics variable
ROWQ	1	@V1	Assign the value in the graphics variable ROW to the BASIC integer variable V1.
RSPR	1	e1	Relocate sprite space by the signed value e1.
SCL	1	e1	Assign the value of e1 to the graphics variable SCL.
SCLQ	1	@V1	Assign the value in the graphics variable SCL to the BASIC integer variable V1.
SCLS	4		Clear the screen and make the current text window Laser BASIC's text window.
SCNP	1	SPN,COL,ROW HGT,LEN,@V1	Scan sprite window for data.
SCNS	1	SPN	Scan whole sprite for data.
SCNV	1	COL,ROW,HGT LEN	Scan screen window for data.
SET	1	e1	Assign the value of e1 to the current set pointer SET.
SETP	5	SPN,COL,ROW HGT,LEN,IK1 IK2	Re-colour the sprite window.
SETQ	1	@V1	Assign the value of the current set pointer to the BASIC integer variable V1.

SETS	5	SPN	Re-colour the whole sprite.
SETV	5	COL,ROW,HGT LEN	Re-colour the screen window.
SP1	1	e1	Assign the value of e1 to the graphics variable SP1.
SP1Q	1	@V1	Assign the value in the graphics variable SP1 to the BASIC integer variable V1.
SP2	1	e1	Assign the value of e1 to the graphics variable SP2.
SP2Q	1	@V1	Assign the value in the graphics variable SP2 to the BASIC integer variable V1.
SP3	1	e1	Assign the value of e1 to the graphics variable SP3.
SP3Q	1	@V1	Assign the value in the graphics variable SP3 to the BASIC integer variable V1.
SP4	1	e1	Assign the value of e1 to the graphics variable SP4.
SP4Q	1	@V1	Assign the value in the graphics variable SP4 to the BASIC integer variable V1.
SPL1	8	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 1 pixel, no wrap.
SPL4	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 1 byte, no wrap.
SPL8	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 2 bytes, no wrap.
SPN	1	e1	Assigns the value of e1 to the graphics variable SPN.
SPNQ	1	@V1	Assigns the value in the graphics variable SPN to the BASIC integer vaiable V1.
SPNV	2	COL,ROW,HGT LEN,SCL,SRW	Spins screen window 90 degrees clockwise.
SPR1	8	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 1 pixel, no wrap.
SPR4	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 1 byte, no wrap.
SPR8	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 1 byte, no wrap.
SPVN	9	SPN,COL,ROW HGT,LEN,NPX e1,e2	Scroll sprite window vertically, NPX pixels, no wrap.
SRW	1	e1	Assign the value of e1 to the graphics variable SRW.
SRWQ	1	@V1	Assign the value in the graphics variable SRW to the BASIC variable V1.
SSL1	8	SPN,e1,e2	Scroll sprite left 1 pixel, no wrap.

SSL4	9	SPN,e1,e2	Scroll sprite left 1 byte, no wrap.
SSL8	9	SPN,e1,e2	Scroll sprite left 2 bytes, no wrap.
SSPR	1	e1,e2	Set sprite space.
SSR1	8	SPN,e1,e2	Scroll sprite right 1 pixel, no wrap.
SSR4	9	SPN,e1,e2	Scroll sprite right 1 byte, no wrap.
SSR8	9	SPN,e1,e2	Scroll sprite right 2 bytes, no wrap.
SSVN	9	SPN,NPX,e1 e2	Scroll sprite vertically NPX pixels, no wrap.
STCP	4	SPN,COL,ROW HGT,LEN,IK1	Set colour throughout sprite window.
STCS	4	SPN	Set colour throughout whole sprite.
STCV	4	COL,ROW,HGT LEN	Set colour throughout whole screen window.
SUMP	1	SPN,COL,ROW HGT,LEN,e1 e2,...,@V1	Sum the sprite window and assign or compare.
SUMS	1	SPN,e1,e2 ,...,@V1	Sum the whole sprite and assign or compare.
SUMV	1	COL,ROW,HGT LEN,e1,e2 ,...,@V1	Sum the screen window and assign or compare.
SVL1	8	COL,ROW,HGT LEN,e1,e2	Scroll the screen window left 1 pixel, no wrap.
SVL4	9	COL,ROW,HGT LEN,e1,e2	Scroll the screen window left 1 byte, no wrap.
SVL8	9	COL,ROW,HGT LEN,e1,e2	Scroll the screen window left 2 bytes, no wrap.
SVR1	8	COL,ROW,HGT LEN,e1,e2	Scroll the screen window right 1 pixel, no wrap.
SVR4	9	COL,ROW,HGT LEN,e1,e2	Scroll the screen window right 1 byte, no wrap.
SVR8	9	COL,ROW,HGT LEN,e1,e2	Scroll the screen window right 2 bytes, no wrap.
SVVN	9	COL,ROW,HGT LEN,NPX,e1 e2	Scroll the screen window vertically NPX pixels, no wrap.
SWPS	4	SP1,SP2,SP3 SP4	Reverse frame sequence.
TMOV	11	KEY,@V1	Execute one instruction in a tracker.
TPUT	1	KEY,SP1,COL ROW,e1	Launch a tracker.
WBNC	11	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2	Bounce using block over-write.
WMOV	11	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,@V1 e1,e2	Linearly move using block over-write.

WMVJ	11	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,KEY @V1,e1,e2	Move under keyboard/joystick control using block over-write.
WPL1	8	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 1 pixel, with wrap.
WPL4	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 1 byte, with wrap.
WPL8	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window left 2 bytes, with wrap.
WPR1	8	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 1 pixel, with wrap.
WPR4	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 1 byte, with wrap.
WPR8	9	SPN,COL,ROW HGT,LEN,e1 e2	Scroll sprite window right 2 bytes, with wrap.
WPVN	9	SPN,COL,ROW HGT,LEN,NPX e1,e2	Scroll sprite window vertically NPX pixels, with wrap.
WSL1	8	SPN,e1,e2	Scroll whole sprite left 1 pixel, with wrap.
WSL4	9	SPN,e1,e2	Scroll whole sprite left 1 byte, with wrap.
WSL8	9	SPN,e1,e2	Scroll whole sprite left 2 bytes, with wrap.
WSR1	8	SPN,e1,e2	Scroll whole sprite right 1 pixel, with wrap.
WSR4	9	SPN,e1,e2	Scroll whole sprite right 1 byte, with wrap.
WSR8	9	SPN,e1,e2	Scroll whole sprite right 2 bytes, with wrap.
WSVN	9	SPN,e1,e2 NPX	Scroll whole sprite vertically NPX pixels, with wrap.
WVL1	8	COL,ROW,HGT LEN,e1,e2	Scroll screen window left 1 pixel, with wrap.
WVL4	9	COL,ROW,HGT LEN,e1,e2	Scroll screen window left 1 byte, with wrap.
WVL8	9	COL,ROW,HGT LEN,e1,e2	Scroll screen window left 2 bytes, with wrap.
WVR1	8	COL,ROW,HGT LEN,e1,e2	Scroll screen window right 1 pixel, with wrap.
WVR4	9	COL,ROW,HGT LEN,e1,e2	Scroll screen window right 1 byte, with wrap.
WVR8	9	COL,ROW,HGT, LEN,e1,e2	Scroll screen window right 2 bytes, with wrap.
XMOV	11	SP1,SP2,SP3 SP4,HGT,LEN COL,ROW,@V1 e1,e2	Linearly move using exclusive-OR.

THE SPRITE GENERATOR
by Cyclone Software

INTRODUCTION

The sprite generator program was developed to compliment the Laser series of languages. The languages are comprised of commands for manipulating sprites and screen data but do not have the facility to directly design sprites. This means there are two phases to games creation. The first involves designing and editing your sprites with the sprite generator program, and the second involves the writing of the game itself using the Laser languages. In practice the two areas of work will probably be carried out simultaneously. The sprite generator program is designed to work in all three of the Amstrad's screen modes. In order to make the sprite generator as easy as possible to use, all operations are executed in the same way regardless of the screen mode you are in. The rotation operation, however, should only be used in mode 1.

THE MAIN MENU

The main menu shows all the options that are available to you. You may select one of these options by pressing the appropriate number. Options 1,2 or 3 will allow you to design sprites. Option 4 allows the sprites you have designed to be saved to tape or disk. Option 5 allows sprites that have previously been designed to be loaded back into the sprite generator. Option 6 allows sprites on tape or disk to be merged with the sprites currently held in the sprite generator. If you are using a disk then option 7 allows you to examine what is on a disk and if you so desire erase a file. Option 8 allows you to animate between several sprites in a defineable pattern. Option 9 allows you to choose between using tape or disk.

GLOSSARY OF TERMS

THE CHARACTER SQUARE

The character square refers to the 8 by 8 grid which is situated in the top left hand corner of the screen. This is the area used to create and edit sprites, one character at a time.

THE SPRITE DISPLAY AREA

This is the larger area located at the bottom of the screen and is the area used to create, develop, transform and generally work on sprites.

THE CHARACTER SQUARE CURSOR

This is the non-destructive flashing cursor which is used to design and edit the character currently held in the character square.

THE SPRITE DISPLAY WINDOW

The area of the screen currently being worked on is refered to as the sprite display window. Its position is defined by SX and SY which corresponds to the position of the top left of the window, and its dimensions are defined by XS and YS. Top left of the sprite display area has co-ordinates SX:0, SY:0. To display the window you are currently working on, press ":" (colon) and it will flash.

SPRITE LIBRARY

This refers to the set of sprites you are currently working with and contains between 1 and 255 sprites. When the program is first RUN you are asked to enter the maximum sprite number which you wish to use.

SPRITES

A sprite is a programmable graphics character. The sprite generator program can develop up to 255 sprites of user selectable dimensions. The amount of free memory in the generator is about 6k, but Laser BASIC can merge a number of sprite files so this is not a problem.

INKS AND PAPER

When working in screen mode 0, 16 INKS are available, in screen mode 1, 4 INKS are available and in screen mode 2 only 2 INKS are available. The colours of these INKS are displayed to the right of the character square. The colour of the INK currently in use is indicated by an arrow. The first INK (INK 0) is referred to as paper and this will be the colour displayed where no pixels are set. Changing the colour of the first INK will cause the whole background of the screen to change colour.

DESIGNING YOUR SPRITES

THE SPACE BAR

The sprite generator program operates in two modes, character mode and sprite mode. In character mode the sprite generator allows you to design characters on the character square and then put down or pick up characters from the sprite display area. In sprite mode the sprite generator allows you to develop and transform your sprites on the sprite display area. The SPACE BAR allows you to switch between character mode and sprite mode.

CHARACTER MODE

Cursor movement

The non-destructive cursor may be moved over the character square using the following keys:

A	to move left;
D	to move right;
W	to move up;
X	to move down.

If the 'SHIFT' key is pressed with one of the above keys then the cursor on the sprite display area will move and CX or CY will be updated. The use of 'A', 'D', 'W' and 'X' is replicated by the joystick, if present.

The 'S' key

This allows you to place a character from the character square onto the sprite display area at the current cursor position which is defined by CX and CY. In order to help you a cursor will flash to indicate the position of CX and CY on the sprite display area.

The 'L' key

This allows you to pick up a character from the sprite display area at the current cursor position and expand it into the character square.

The 'I' key

Pressing the 'I' key will cause the INK currently in use to cycle through all the available colours until the 'I' key is released.

The 'C' key

Pressing the 'C' key will simply clear the character square. If the 'SHIFT' key is pressed at the same time then the character square will remain intact but the sprite display area will be cleared.

The SPACE BAR

Pressing the SPACE BAR puts the sprite generator into sprite mode.

The 'P' key

Pressing the 'P' key will move the arrow indicating the INK being used downward, if the 'SHIFT' key is pressed at the same time then the arrow will move upward.

The 'J' key

When sprites are used in a program the colour of the INKs will usually be set to those used when the sprites were designed. To make this easier pressing the 'J' key will display the colour number that all the INKs are set to.

The 'B' key

Pressing the 'B' key changes the colour of the border.

The 'CLR' key

Pressing the 'CLR' key homes both of the cursors (sends them back to the top left).

The 'ENTER' key

The 'ENTER' key sets the pixel at the current cursor position on the character square grid to the current pen. If a joystick is connected then the fire button can be used to carry out the same function.

The 'DEL' key

The 'DEL' key sets the pixel at the current cursor position on the character square to the paper colour (INK 0).

The 'E' key

Pressing the 'E' key allows you to enter data directly onto the sprite display area at the current position. When the 'E' key is pressed you will be asked whether you want to enter the data in decimal or binary. You should respond by pressing either 'D' or 'B' respectively. You will then be asked to enter 8 bytes of data. These will be placed directly onto the sprite display area and will leave the character square unaffected.

The 'R' key

Pressing the 'R' key with the 'SHIFT' key will return you to the main menu.

SPRITE MODE

Sprite display window movement

The sprite display window may be moved using the following keys:

A	to move left;
D	to move right;
W	to move up;
X	to move down.

Sprite display window size

The size of the sprite display window may be altered by pressing the following keys together with the 'SHIFT' key.

A	to decrease length of window;
D	to increase length of window;
W	to decrease height of window;
X	to increase height of window.

Note that the use of 'A', 'D', 'W' and 'X' to move and re-dimension is replicated by the joystick if connected.

The '@' key

Increases the X-increment (in bytes) by which the sprite display window will move. The increment will increase up to a value of 8, then re-start from 1.

The ';' key

Increases the Y-increment (in pixels) by which the sprite display window will move. Again the increment will increase until it reaches a value of 8 and then re-start from 1.

The 'L' key

Pressing the 'L' key allows a previously created sprite to be placed on the sprite display area at the current cursor position. The whole sprite will be put onto the sprite display area regardless of the current size of the sprite window.

The 'S' key

This allows you to 'GET' a sprite into memory. Pressing the 'S' key creates a sprite with the dimensions of the current display window and the current SPN value (provided it has not already been allocated). The data in the window is automatically 'GOT' into the sprite. If the sprite number were previously allocated then a low 'beep' would be issued and no other action taken. If the operation were successful then a high 'beep' would be heard.

The 'P' key

The sprite generator program puts sprites to the sprite display area in 1 of 6 ways. This is referred to as the print mode. The six print modes are as follows and are represented by PMD.

BL	Block 'PUT' data onto sprite display area
OR	OR data onto sprite display area
XOR	XOR data onto sprite display area
AND	AND data onto sprite display area
IF	Place data in front of the data currently on the sprite display area
BH	Place data behind the data currently on the sprite display area

Pressing the 'P' key will cycle through these print modes.

The 'N' key

Pressing the 'N' key will increment the current sprite number (represented by SPN). If the 'SHIFT' key is pressed at the same time as the 'N' key then the current sprite number will be decremented.

The 'F' key

Switches on the special FILL cursor which can be guided around the sprite display area by the 'A','D','W' and 'X' keys (or a joystick). Pressing 'ENTER' or 'FIRE' will FILL the appropriate area, from the cursor position, with the current INK. Pressing SHIFT and 'F' will set the whole sprite display window to the current INK colour.

The 'E' key

The 'E' key allows a previously created sprite to be erased. Before actually erasing the sprite you will be asked "ARE YOU SURE? (Y/N)" this is to safeguard against the accidental erasure of sprites.

The 'O' key

Pressing the 'O' key toggles 'wrap' for the scrolls (represented by WRP) on and off. If 'wrap' is turned on when a scroll is performed any data that goes off one side of the sprite display window reappears on the other side, whereas if 'wrap' is turned off it does not.

The Arrow keys

The arrow keys have two functions. When used on their own the sprite display window is scrolled in the appropriate direction. If, however, the 'SHIFT' key is pressed at the same time then a flip (mirroring) will be performed. The following types of flips can be achieved by pressing the 'SHIFT' key with the arrow keys:

← flips the contents of the sprite display window about the Y axis.

↑ flips the contents of the sprite display window about the X axis

→ flips the contents of the sprite display window to the right of the window (creates a horizontally symmetric window).

↓ flips the contents of the sprite display window to the bottom of the window (creates a vertically symmetric window).

Using the '→' and '↓' keys it is possible to produce a symmetrical design whilst only having to create half or a quarter of the full design.

The 'B' key

If the 'B' key is pressed then the contents of the sprite display window will be magnified horizontally.

If the 'SHIFT' key is used in conjunction with the 'B' key then the contents of the sprite display window will be magnified vertically.

The 'T' key

Using the 'T' key you are able to rotate the contents of the sprite display window by 90 degrees in a clockwise direction into another area of the screen. When the 'T' key is pressed it is taken that the data contained within the sprite display window is the data to be rotated. The sprite display window should be placed in the position where the data is to be rotated to. When the cursor is in position the 'ENTER' key should be pressed. The area to which the data is being rotated should not overlap the area from which the data is taken. The height of the area to be rotated is rounded down to the nearest multiple of 8 pixels. See SPNV.

The 'Z' key

Pressing the 'Z' key will invert the contents of the sprite display window.

The 'H', 'G' and 'V' keys

It may be necessary at some stage to design a sprite that is bigger than the size of the sprite display area. This may be done by creating a large sprite in memory and then designing it, a section at a time. Pressing the 'H' key will create a large sprite in memory. You will be asked to enter the width of the sprite in bytes and the height of the sprite in pixels. Pressing the 'G' key 'GETs' the contents of the sprite display window into the large sprite. You will be asked for the column and row positions within the sprite. Pressing the 'V' key will place a section of a large sprite onto the sprite display area. The position at which this is placed and the size of the section to be placed will be indicated by the sprite display window. You will be asked to enter the row and column position within the sprite from which the data is to be taken. The sprite number is again indicated by SPN.

The '[' key

Pressing the '[' key changes the amount by which the sprite display window will be scrolled when a horizontal scroll is performed.

The ']' key

Pressing the ']' key changes the amount by which the sprite display window will be scrolled when a vertical scroll is performed.

The 'K' key

This option provides you with the facility to change the colour of particular pixels in the sprite display window for some other colour (see SETV). This does not effect the colour of the INKs but actually changes the pixels in the sprite display area. When the 'K' key is pressed you will be asked for the first INK. This is the INK number of the pixels to be changed. Then you will be asked to enter the second INK. This is the INK number to which the pixels are to be changed.

The 'M' key

The 'M' key performs the function of masking and de-masking a sprite. If the 'M' key is pressed then the sprite who's number is shown in SPN will be masked. If the 'SHIFT' key is pressed at the same time as the 'M' key then the sprite who's number is shown in SPN will be de-masked.

The ':'

When the ':' key is pressed, the current sprite display wondow will flash until the key is released. This option is provided to locate the position and size of the current window.

The 'I' key

When the 'I' key is pressed a new set of information will be shown.

The 'R' key

Pressing the 'SHIFT' key at the same time as the 'R' key will return you to the main menu. Please note all data contained in both the character square and in the sprite display area will be lost when you return to the main menu. All system pointers are also reset to their default values.

ANIMATING SPRITES

You are able to animate between several sprites by choosing option 8 from the main menu. The sequence through which the sprite generator will animate is entered at the top of the screen, the actual animation will take place on the sprite display area at the bottom.

Entering an animated pattern

To enter a sprite into the animation sequence you must move the cursor to the position in the sequence where the sprite is to be entered. Then type in the number of the sprite that you wish to enter at the position followed by 'ENTER'. At the end of your animation pattern enter 'R' and at this point the animation pattern will start from the beginning again.

Cursor movement

The cursor that indicates the current position in the animation sequence may be moved by the following keys:

- **A** to move left;
- **D** to move right;
- **W** to move up;
- **X** to move down.

If the 'SHIFT' key is pressed with one of the above keys then the cursor on the sprite display will move. This cursor indicates the position at which the animation sequence will be displayed.

The SPACE BAR

Pressing the SPACE BAR starts and stops the animation sequence.

The 'R' key

Pressing the 'SHIFT' key at the same time as the 'R' key will return you to the main menu.

SAVING SPRITES

You can save your sprites onto tape or disk by selecting option 4 from the main menu. After selecting this option you will be asked for the filename under which the sprites are to be saved. The filename should be typed in upper case with a maximum of 5 characters. Three sets of data are saved, the first contains system variables used by the Laser routines, the second contains the sprite table holding information about each sprite, and the third is the actual sprite data.

LOADING SPRITES

Option 5 will allow a file of previously saved sprites to be loaded from tape or disk. After selecting this option you will be asked for the filename of the sprites to be loaded. "SPR" will automatically be appended to the filename you enter. The new maximum sprite number (from the loaded file) is displayed and an option is given to alter it. If you respond with "Y" to the prompt then a new SMAX should be entered. This must be in the range 1 to 255. If the file is not found, the prompts will still be issued but you will probably respond with "N" unless you wish to modify SMAX for some other reason.

MERGING SPRITES

Sprite files can also be merged from tape or disk. However, this can only be done if none of the sprites being merged from tape or disk have the same number as any of the sprites that are currently held in the sprite generator and if the maximum number of the sprites being merged does not exceed the current maximum sprite number, entered at the start of the session when you first RAN the program. Again, after selecting to merge sprites you will be asked to enter the filename of the sprites to be merged from tape or disk.

A SAMPLE SESSION WITH THE SPRITE GENERATOR

If you haven't already loaded Laser BASIC then load this first. The sprite generator program can now be loaded and RUN, making sure the keyboard is in upper case before RUNning - if not press CAPS SHIFT. You should also ensure that the volume is turned up to make error 'beeps' audible.

Tape users place Tape 2 Side A in the cassette and type RUN"SPTGEN. Disk users just type RUN"SPTGEN to load and run the sprite generator program.

When the program first executes you will be prompted to enter the maximum sprite number. This should be a value in the range 1 to 255, but for now, enter 120. The message "IF YOU HAVE MADE AN ERROR YOU WILL HEAR ..." will appear, and two low beeps will be heard. This is to demonstrate what will be heard when an error occurs. The message "IF THE OPERATION WAS SUCCESSFUL ..." will appear and two high beeps will be heard. This is to demonstrate what will be heard if an operation is successful. Now press any key and the main menu will appear.

You will now need some sprites to work with; we will use the SPT1 sprites (see Appendix A).

The SPT1 sprites should be loaded using the following procedure:

1. If you are using tape then you should place the cassette containing the SPT1 sprites into the cassette player. The SPT1 sprites are situated directly after the sprite generator on tape 2 side A.

2. From the main menu select option 5. This allows you to load your sprites.
3. The prompt "ENTER FILENAME" should appear. You should now enter "SPT1" followed by the key marked 'ENTER'. Note that "SPR" is automatically appended to the filename. For the purposes of this session, do not modify the maximum sprite number (so type N).
4. You will now return to the main menu.

We are going to start the sample session in 16 colour mode (MODE 0) so select option 1 from the main menu.

THE CURSOR KEYS

The 'A','D', 'W' and 'X' keys will allow you to move the flashing cursor around the character square. By pressing the 'SHIFT' key and the above keys you may move the cursor around the sprite display area. Use the keys to move both of them around until you get a feel for it. Notice that the cursors will wrap around, that is to say they will re-appear on the opposite side if they are moved off the edge of the character square or the sprite display area.

In order to set a particular pixel, move the character square cursor to the required position, release the cursor keys and press 'ENTER'. Move the cursor to the next position you want to set and press 'ENTER' again. To unset a pixel, position the cursor over the set pixel and press 'DEL'. If neither of these keys are pressed then the cursor moves non-destructively. That is to say it moves around the screen without affecting the cells it moves across. Now spend a few minutes getting used to the cursor keys by creating, for example, a space invader. It doesn't have to be a work of art, but will serve to demonstrate some of the package's functions.

The 'S' and 'L' Keys

Now that you have designed a character it's time to see what it will look like, reduced to real size on the sprite display area. Press 'S' to put your invader onto the sprite display area, it will appear at the current sprite display area cursor position. Now press 'C' and this will clear the character square. Press 'L' to lift the invader back to the character square. Using this method, sprites can be created or edited a section at a time.

Before going any further, let's take a quick look at the internal format of the Amstrad's pixel data. You may skip this section if you wish and come back to it at a later date.

When characters are stored on the screen they are stored either in 8 bytes, 16 bytes or 32 bytes, depending on which screen mode you are in at the time. A byte is an 8 bit number. The bits are numbered from 0 to 7 starting from the right hand side, and each bit represents the value of two to the power of the number of the bit. Therefore, if just bits 2 and 5 were set, then the value of the byte would be 36. The corresponding values of each bit in a byte are shown below in Fig. 1. Bit 0 is the rightmost bit, bit 7 is the leftmost bit.

```
FIG.1  Bit 0 = 2 to the power of 0 =   1
       Bit 1 = 2 to the power of 1 =   2
       Bit 2 = 2 to the power of 2 =   4
       Bit 3 = 2 to the power of 3 =   8
       Bit 4 = 2 to the power of 4 =  16
       Bit 5 = 2 to the power of 5 =  32
       Bit 6 = 2 to the power of 6 =  64
       Bit 7 = 2 to the power of 7 = 128
```

In screen mode 0, 32 bytes are needed to produce a character. The first byte produces the colour of the first two pixels in the top line of the character, the second byte produces the second two pixels in the top line of the character, the third byte produces the next two pixels, the fourth byte the last two pixels, then the fifth byte will produce the colour of the first two pixels of the second row of the character. Each byte represents 2 pixels which are encoded as follows:

Right pixel - bits 0,4,2,6
Left pixel - bits 1,5,3,7

In screen mode one, 16 bytes are required to store a character. The first byte produces the colour of the first four pixels in the top row of the character. The second byte produces the colour of the last four pixels in the top row of the character, the third byte produces the colour of the first four pixels in the second row of the character and so on. Each byte represents 4 pixels which are encoded as follows:

Left pixel - bits 0,4
Second pixel - bits 1,5
Third pixel - bits 2,6
Right pixel - bits 3,7

In screen mode two, only eight bytes are required to store a character. In this screen mode each bit simply indicates whether the corresponding pixel in the character row is on or off. Now for our invader, firstly press the 'E' key then respond to the question "Decimal or Binary" by pressing the 'D' key. You will then be asked to enter the first byte value which can be found in Fig. 2. Then enter the second byte, third byte and so on until all 8 are entered. Now move the cursor on the sprite display area one place to the right and enter the 8 bytes in Fig. 3, using the same procedure used to enter the data in Fig. 2. Then move the cursor another place to the right and enter the 8 bytes in Fig. 4. Then move the cursor another place to the right and enter the last 8 bytes in Fig. 5.

	Fig.2	Fig.3	Fig.4	Fig.5
1st byte =	17	0	0	34
2nd byte =	34	0	0	17
3rd byte =	17	51	51	34
4th byte =	51	17	34	51
5th byte =	51	17	34	51
6th byte =	51	51	51	51
7th byte =	17	34	17	34
8th byte =	34	0	0	17

Back to the Sample Session....

Now let us change the INK colours. Press the 'P' key and you will cause the arrow to move down the INKs (SHIFT 'P' moves the arrow up). If you now press 'ENTER', a pixel on the character square is set to the INK colour pointed to by the arrow - try it. Although there are only 16 INKs available in this mode, by pressing the 'I' key you will cause the current INK to cycle through all of the available colours. You will notice that all the pixels on both the character square and the sprite display area of the INK pointed to by the arrow will change colour. If you now press the 'J' key, all the values of the INKs will be displayed, and these should be noted down before saving sprites, so that you can set the INKs in your Laser BASIC programs to the required values. Press 'J' again to return to the original screen.

Now press the 'SPACE BAR' to enter sprite mode. Notice that the cursor indicators (CX and CY) have been replaced by the sprite display window indicators (SX and SY). To view the window, hold down ':' and it will flash to indicate its size and position. The window can be moved by pressing the 'A', 'D', 'W' and 'X' keys (to move left, right, up and down respectively) or by using a joystick. The width of the window can be increased or decreased in length by pressing SHIFTed 'A' or SHIFTed 'D' and the height can be increased or decreased by pressing SHIFTed 'W' or SHIFTed 'X'. Again, SHIFT can be used together with the joystick to produce the same result.

CREATING A SPRITE

Move the window to the top left hand corner of the sprite display area so that the SX and SY values indicate 0. Now press the 'I' key and a new set of information will be displayed. Press the 'N' key (or SHIFT 'N' key) until the value of SPN in the top right hand corner is 5. Now press 'L' and sprite 5 will appear. Press 'I' again and another set of information will be displayed, giving all the information about sprite 5. Press 'I' again and you will be back to the original information.

We will pretend this sprite has just been designed. You could if you wish go back to character mode ('SPACE BAR') and change the colours of the INKs; if you do, press the 'SPACE BAR' to return to sprite mode.

The following sequence must be followed to create a sprite. Adjust the sprite window until it contains all the pixel data required in the sprite. You will notice that the values of LEN and HGT will change; in this example LEN should be around 7 and HGT around 23. Press 'I' and set SPN to 26 (an undefined sprite).

Press 'S' (you will hear a high beep) and the sprite will be created. Press 'I' and all the information for the sprite will be displayed. Press 'I' again to return to the original information display. Move the sprite window to a clear part of the screen. Press 'L' and if all has gone well, the new sprite will be put on the screen.

NOTE: If you try to create a sprite with a number which has already been allocated, then a low pitched beep will be heard. If the latter is ever the case then try again using a different SPN number.

You may also erase a sprite by setting SPN to the appropriate value and then pressing the 'E' key. The number of the sprite that you have just erased can now be re-allocated.

The SPT1 sprites you have loaded into memory are a mixture of both 16 and 4 colour mode sprites but only 16 colour mode sprites will be displayed correctly in the current mode (main menu option 1).

We will now display another one of the SPT1 sprites. First decrease SPN to 6 using 'SHIFT' and 'N', then press 'L' and sprite 6 will appear. Move the sprite display window with the usual keys 'A', 'D', 'W' and 'X' until its top left hand corner aligns with the displayed sprite's top left hand corner. Now modify the size of the sprite display window using 'SHIFT' and 'A', 'D', 'W' and 'X' until it fits over the sprite.

SPRITE SCREEN TRANSFORMATIONS

We will now perform some scrolls and flips. Press the right arrow key and the sprite will scroll to the right. Notice that WRAP is off at the moment (WRP is set to 0); if WRAP was on, then the part of the sprite that had disappeared would wrap around to the other side. Now press the downwards arrow key and the sprite will scroll towards the bottom, the WRAP is still off. Using 'L', place the sprite back onto the display area. Press the 'O' key and WRAP will be on, (WRP is set to 1) and now see the effect of scrolling the sprite. Press 'O' again to set WRP to 0.

Now let's perform a flip. Press 'SHIFT' and 'UP ARROW', and the contents of the sprite display window will be turned upside down (vertically mirrored).

We are now going to FILL an area with a particular INK. For this example you will need to design an enclosed outline on the sprite display area. Once you have done this, change the INK and enter sprite mode. Then press 'F'. You will now have control of a small flashing pixel. Using keys 'A', 'D', 'W' and 'X' or the joystick, move this flashing pixel inside the outline and press 'ENTER' or 'FIRE', and the whole interior should be FILLed with the current INK. If there were any gaps in your outline, the INK will be seen to flood all over the sprite display area!!

Move the sprite display window to a free part of the sprite display area, enter character mode and move the arrow to a particular INK. Now go back into sprite mode ('SPACE BAR') then press 'SHIFT' and 'F'. The whole of the sprite window will be flooded with the INK you chose.

Press the 'L' key to put sprite 6 on the screen. Position the window over the sprite.

Now let's change all the pixels in the sprite display window which have a particular INK value to have a different INK value. Press 'K' and you will be prompted to input the INK number you wish to change, so type in number 3 and then 'ENTER'. You will now be asked to enter the INK number you wish to change it to, so type 14 and then 'ENTER'. This will change everything drawn in INK 3 to INK 14.

At this stage let's go back to the main menu. Go back to character mode by pressing the 'SPACE BAR'. Before you return to the main menu check that all the sprites on the sprite display area have been stored in memory as the sprite display area is about to be cleared. Now press 'SHIFT' and 'R' and the main menu will be displayed.

MODE 1 SPRITES

Select Option 2 to use the sprite generator in graphics mode 1. The display is similar to the one selected by option 1.

Press the SPACE BAR and then press 'L'. SPN will be set to 1 (the default value on selecting the option) so sprite 1 will appear on the screen (sprite 1 is a MODE 1 sprite). You may, if you wish, go back to character mode and set the INKs to their preferred colours.

ROTATION

We are now going to demonstrate rotation (which will only work correctly in this mode). Firstly set the sprite display window to cover the tank (remember that HGT must be divisible by 8). LEN should be 10 and HGT should be 24.

You can check that the sprite display window lies over the tank by holding down ':' or by pressing 'Z' to invert the window. If you do invert the window by pressing 'Z' then 'Z' should be pressed again before you proceed.

Press 'T' and move the sprite display area window to a free part of the screen (clear of the tank) and then press ENTER. A 90 degree rotated tank will appear.

SPRITE DISPLAY WINDOW AND SPRITE CURSOR REVISITED

By now you will probably have noticed that it takes a long time to move the sprite display window or cursor around the sprite display area. The sprite generator is provided with a facility to change the X and Y increments which define the speed the display window and cursor move at. In both character mode and sprite mode, the '@' key will change the value of XS, and the ';' key the value of YS. The values for the two modes are, however, independant, so changing XS and YS in character mode will not affect the values displayed when you enter sprite mode. Press the '@' key until XS is 8. Press the ';' key to change the value of YS until it is also 8. You can now move around in large steps.

The rate of sprite display window scrolling can be changed using the '[' and ']' keys. In sprite mode press the 'I' key. You will see two values XSC and YSC displayed. XSC is the resolution of sideways scrolling. This is set to 'PIX' when the sprite generator is first loaded. This gives the scrolling a resolution of 1 pixel.

If you now press '[' a resolution of 1 byte will be set, press '[' again and a 2 byte resolution is set. Press '[' and try a sideways scroll.

YSC indicates the vertical scrolling resolution. Pressing '[' will increase the value in a cycle of 1 to 8. Try some scrolls with various values of XSC and YSC to see the results.

MORE SCREEN TRANSFORMATIONS

Place the sprite display window at the top left of the sprite display area, i.e. SX and SY should be set to 0. Press 'L' to put sprite 1 onto the screen. Now move the sprite display window to cover the sprite (a tank). Check its position using either the ':' key or the 'Z' key.

Press the 'B' key and the tank is magnified by a factor of 2 horizontally. You will notice that the LEN value is now twice the original value.

Press 'SHIFT' and 'B' and the tank is magnified by a factor of 2 vertically. HGT now has twice its original value. If magnification would produce a result larger than the dimensions of the sprite display area then no action is taken.

ANIMATION

Most arcade games require a sequence of animated sprites. The sprite generator program provides an option which enables you to view an animation sequence.

Return to the main menu by going into character mode and then pressing SHIFT 'R'.

There are lots of examples of sprite animation in the demo, so we will now load in the SPT3 sprites.

Tape users will have to insert the demo tape.

Select option 5 and type SPT3. Press 'N' in response to 'change sprite max value', and once you have returned to the main menu select option 8.

You will see that a cursor ('>') is positioned at the top left hand corner of the screen. The sprites we are going to animate are sprites 30 to 33 (the 'eyeballs').

Type in 30 followed by ENTER, now move the cursor to position 2 using the 'D' key and type in 31 followed by ENTER. Move the cursor to position 3 using the 'A' and then the 'X' key and type in 32. Finally, enter 33 at position 4.

Move the cursor to position 5 and press 'R'.

The animation data has now been entered and is ready to be run. Press the 'SPACE BAR' to animate.

To return to the main menu type SHIFT 'R'.

MORE ADVANCED FEATURES

Return to the main menu and select option 2.

MASKING AND REMASKING

Sprites that are going to be moved using, for example, FMOV or BMOV must be MASKed. You can do this in your Laser BASIC program or in the sprite generator program.

Set SPN to 10 and press 'L' to put sprite 10 onto the sprite display area. The first thing you will notice is a series of vertical bars indicating that the sprite is already MASKed. Press SHIFT 'M' and you should hear a high beep meaning the operation was successful. What you have done is de-MASK the sprite in memory. Press 'L' to rePUT the sprite and see the result.

To MASK a sprite you must make sure, as in sprite 10, that all the useful data is in the left hand side of the sprite. Press 'M' and a high beep should signify a successful operation. Now press 'L' and the MASKed sprite will be displayed.

NUMERICAL DATA ENTRY

As stated earlier in this manual, sprites can be used to hold data for various applications. If you press the 'SPACE BAR' to go back into character mode, and then press 'E', 'DECIMAL OR BINARY' is displayed. Press 'D' for DECIMAL and 'BYTE 1' is displayed, so type in 1 and hit ENTER followed by 2, 4, 8, 16, 32, 64 and 128.

The data you have just entered will appear at the sprite screen cursor position.

SAVING YOUR FINISHED SPRITES

Once you have created your sprites, go back to the main menu. Toggle option 9 to point to either 'DISK' or 'TAPE' depending on your system. This is acheived by pressing the '9' key.

Now press '4' to select option 4 and input the filename. Remember you do not put 'SPR' at the end of the filename as this is done for you, e.g. ENTER 'FRED' and a file of sprites named 'FREDSPR' will be saved.

If you accidentally break out of the sprite generator program, save off the sprites manually using PSPR before typing RUN. Running the program will erase all the sprites from memory.

FUNCTION KEY SUMMARY

MAIN MENU OPTIONS:

OPTIONS	**FUNCTIONS**
1,2,3	Allows you to design your sprites.
Character	Mode
KEYS	
A	To move left.
D	To move right.
W	To move up.
X	To move down.
SHIFT with (A,D,W,X)	Will move cursor around sprite display area.
S	Places a character to the sprite display area.
L	Picks up a character from the sprite display area.
I	Changes the colour of the current INK.
C	Clears the character square.
SHIFT and C	Clears the sprite display window.
SPACE BAR	Puts the sprite generator program in to sprite mode.
P	Moves arrow indicating the INK down.
SHIFT and P	Moves arrow indicating the INK up.
J	Displays the colour value of all the INKs.
B	Changes the colour of the border.
CLR	Homes both of the cursors.
ENTER	Sets a pixel to the current INK.
DEL	Sets a pixel to the paper colour.
E	Enters data on to the sprite display area.
R and SHIFT	Returns you to the main menu.

Sprite Mode

KEYS	
A	To move left.
D	To move right.
W	To move up.
X	To move down.
SHIFT and A	To decrease length of window.
D	To increase length of window.
W	To decrease height of window.
X	To increase height of window.
@	Changes the X-increment for the sprite display window movement.
;	Changes the Y-increment for the sprite display window movement.
L	Allows a sprite to be placed on the sprite display area.
S	Allows a sprite to be saved to memory.
P	Changes print mode.
N	Increases current sprite number.
SHIFT and N	Decreases current sprite number.
F	Fills an area bounded by pixels.
SHIFT and F	Floods the sprite display window with the current pen.
SHIFT and E	Erases a sprite.
O	Toggles wrap on and off.
↓	Scrolls down.
→	Scrolls right.
←	Scrolls left.
↑	Scrolls up.
SHIFT and ←	Flips contents of window about the Y axis.
SHIFT and ↑	Flips contents of window about the X axis.
SHIFT and →	Flips contents of window to the right of the window.
SHIFT and ↓	Flips contents of window to the bottom of the window.
B	Magnifies window horizontally.
SHIFT and B	Magnifies window vertically.
T	Rotates window by 90 degrees.
Z	Invert the window contents.
H	Creates a large sprite in memory.
G	Saves the contents of the window to the large sprite.
V	Loads a section of a large sprite from memory to the sprite display area.
[	Changes the step when performing a horizontal scroll.
]	Changes the step when performing a vertical scroll.
K	Changes all occurrences of a INK in the sprite display window for some other colour.
M	Masks a sprite.
SHIFT and M	De-masks a sprite.
:	Flash the current sprite display window.
I	Displays information.
SHIFT and R	Returns to the main menu.

Animating your Sprites

OPTION	FUNCTION
8	
KEYS	
A	To move left.
D	To move right.
W	To move up.
X	To move down.
SHIFT and (A,D,W,X)	Moves cursor on sprite display area.
SPACEBAR	Starts and stops animation sequence.
SHIFT and R	Returns to main menu.

APPENDIX A — SPT1 SPRITES

To load sprites into Laser BASIC, type: `A$="SPT1SPR":|GSPR,@A$`

Sprite No.	Description			Suggested colour values for inks													
		HGT	LEN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Light tank	19	9	0	13	12	–	–	–	–	–	–	–	–	–	–	–	–
2	WWII Zero fighter	13	8	0	23	6	–	–	–	–	–	–	–	–	–	–	–	–
3	WWII Stuka divebomber	16	8	0	9	11	–	–	–	–	–	–	–	–	–	–	–	–
4	Army helicopter	15	11	0	9	13	–	–	–	–	–	–	–	–	–	–	–	–
5	Ghost	22	7	0	18	9	14	6	0	0	0	0	0	0	0	0	0	0
6	Mutant strawberry	20	5	0	16	6	26	3	9	18	2	0	0	0	0	0	0	0
7	Mutant orange	16	6	0	14	15	26	3	9	0	0	0	0	0	0	0	0	0
8	Space cruiser	19	10	0	25	15	–	–	–	–	–	–	–	–	–	–	–	–
9	Space scout–ship	19	9	0	14	23	–	–	–	–	–	–	–	–	–	–	–	–
10	Space transporter	14	11	0	0	8	–	–	–	–	–	–	–	–	–	–	–	–
11	Deep sea submarine	18	15	0	24	15	–	–	–	–	–	–	–	–	–	–	–	–
12	Lunar rover	15	10	0	10	15	–	–	–	–	–	–	–	–	–	–	–	–
13	Lunar transporter	18	14	0	13	15	–	–	–	–	–	–	–	–	–	–	–	–
14	Jet interpreter	16	13	0	12	13	–	–	–	–	–	–	–	–	–	–	–	–
15	Alien	22	6	0	2	11	26	6	–	–	–	–	–	–	–	–	–	–
16	Sports car	15	13	0	2	11	–	–	–	–	–	–	–	–	–	–	–	–
17	Lunar explorer	21	10	0	5	12	–	–	–	–	–	–	–	–	–	–	–	–
18	Martian explorer	27	16	0	13	14	–	–	–	–	–	–	–	–	–	–	–	–
19	Martian probe	35	10	0	3	24	–	–	–	–	–	–	–	–	–	–	–	–
20	Helicopter transporter	19	15	0	9	12	–	–	–	–	–	–	–	–	–	–	–	–
21	Helicopter gunship	17	15	0	12	11	–	–	–	–	–	–	–	–	–	–	–	–
22	Jet helicopter	16	15	0	13	11	–	–	–	–	–	–	–	–	–	–	–	–
23	Jet fighter	16	13	0	9	12	–	–	–	–	–	–	–	–	–	–	–	–
24	Saloon car	16	14	0	6	13	–	–	–	–	–	–	–	–	–	–	–	–
25	Shuttle	20	13	0	26	13	–	–	–	–	–	–	–	–	–	–	–	–

APPENDIX B — SPT2 SPRITES

```
To load sprites into Laser BASIC, type: A$="SPT2SPR":|GSPR,@A$
```

Sprite No.	Description			Suggested colour values for inks														
		HGT	LEN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Spider	16	4	14	6	0	–	–	–	–	–	–	–	–	–	–	–	–
2	Droid	16	4	15	18	0	–	–	–	–	–	–	–	–	–	–	–	–
3	Heart	16	8	0	0	6	–	–	–	–	–	–	–	–	–	–	–	–
4	<defined in example session>	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
5	<defined in example session>	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
6	OCEAN IQ Logo	32	28	2	26	11	–	–	–	–	–	–	–	–	–	–	–	–
7	Ball	8	2	18	0	9	–	–	–	–	–	–	–	–	–	–	–	–
8	Joystick	16	4	6	0	13	–	–	–	–	–	–	–	–	–	–	–	–
9	Face	16	4	16	15	0	–	–	–	–	–	–	–	–	–	–	–	–
10	Teddy bear	40	8	15	17	0	–	–	–	–	–	–	–	–	–	–	–	–
11	Little man	16	4	16	7	9	–	–	–	–	–	–	–	–	–	–	–	–
12	Vintage car	16	16	0	13	14	–	–	–	–	–	–	–	–	–	–	–	–
13	Dragster	16	16	0	15	18	–	–	–	–	–	–	–	–	–	–	–	–
14	Bi-plane	16	3	13	0	0	–	–	–	–	–	–	–	–	–	–	–	–
15	Dummy sprite	18	9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
16	<defined in example session>	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
17	Mutant plant #1	22	7	19	0	9	–	–	–	–	–	–	–	–	–	–	–	–
18	Mutant plant #2	22	7	19	0	9	–	–	–	–	–	–	–	–	–	–	–	–
19	Mutant plant #3	22	7	19	0	9	–	–	–	–	–	–	–	–	–	–	–	–
20	Mutant plant #4	22	7	19	0	9	–	–	–	–	–	–	–	–	–	–	–	–
21	Walking robot #1	16	6	0	26	10	–	–	–	–	–	–	–	–	–	–	–	–
22	Walking robot #2	16	6	0	26	10	–	–	–	–	–	–	–	–	–	–	–	–
23	Walking robot #3	16	6	0	26	10	–	–	–	–	–	–	–	–	–	–	–	–
24	Walking robot #4	16	6	0	26	10	–	–	–	–	–	–	–	–	–	–	–	–
25	Scissors #1	20	8	13	0	0	–	–	–	–	–	–	–	–	–	–	–	–
26	Scissors #2	20	8	13	0	0	–	–	–	–	–	–	–	–	–	–	–	–
27	Scissors #3	20	8	13	0	0	–	–	–	–	–	–	–	–	–	–	–	–
28	Scissors #4	20	8	13	0	0	–	–	–	–	–	–	–	–	–	–	–	–
29	Undefined	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
30	<defined in example session>	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
31	Bricks	25	10	0	0	6	–	–	–	–	–	–	–	–	–	–	–	–
32	Dummy sprite	25	10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
33	The letter 'A'	8	2	24	0	0	–	–	–	–	–	–	–	–	–	–	–	–
34	Dummy sprite	8	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
35	The letter 'a'	8	2	24	0	0	–	–	–	–	–	–	–	–	–	–	–	–
36	Dummy sprite	2	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
37	The letter 'a'	8	2	24	0	0	–	–	–	–	–	–	–	–	–	–	–	–
38	Blank block	8	2	24	0	0	–	–	–	–	–	–	–	–	–	–	–	–
39	Banana	17	5	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	Data sprite	1	13	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
41	Data sprite	1	11	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
42	Data sprite	1	170	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
43	Data sprite	1	170	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
44	Data sprite	1	19	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
45	Data sprite	1	7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
46	Planet with moon #1	16	4	11	26	6	–	–	–	–	–	–	–	–	–	–	–	–
47	Planet with moon #2	16	4	11	26	6	–	–	–	–	–	–	–	–	–	–	–	–
48	Planet with moon #3	16	4	11	26	6	–	–	–	–	–	–	–	–	–	–	–	–
49	Planet with moon #4	16	4	11	26	6	–	–	–	–	–	–	–	–	–	–	–	–
50	Site	13	3	23	0	0	–	–	–	–	–	–	–	–	–	–	–	–
51	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
52	Arrowed square	24	6	25	6	0	–	–	–	–	–	–	–	–	–	–	–	–
53	Cat	22	7	0	12	0	–	–	–	–	–	–	–	–	–	–	–	–
54	Bird	25	5	0	0	4	–	–	–	–	–	–	–	–	–	–	–	–

APPENDIX C(I) — SPT3 SPRITES (Sprites used in the demo part 1)

To load sprites into Laser BASIC, type: A$="SPT3SPR":|GSPR,@A$

Please note that all sprites postfixed with (M) are in a MASKed form and can be demasked using DMSK.

Sprite No.	Description			Suggested alternative colours for the inks													
		HGT	LEN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Tortoise	17	8	0	6	12	–	–	–	–	–	–	–	–	–	–	–	–
2	Rat	17	9	0	6	10	–	–	–	–	–	–	–	–	–	–	–	–
3	Hare	17	8	0	20	15	–	–	–	–	–	–	–	–	–	–	–	–
4	Stone floor section	8	7	0	11	9	–	–	–	–	–	–	–	–	–	–	–	–
5	Rope and pulley lift	23	7	0	11	7	–	–	–	–	–	–	–	–	–	–	–	–
7	Cube	8	2	0	16	20	–	–	–	–	–	–	–	–	–	–	–	–
8	Space scout craft	23	6	0	10	16	–	–	–	–	–	–	–	–	–	–	–	–
9	Alien	24	10	0	8	0	–	–	–	–	–	–	–	–	–	–	–	–
10	Hopping alien #1(M)	33	10	0	8	0	–	–	–	–	–	–	–	–	–	–	–	–
11	Hopping alien #2(M)	33	10	0	8	0	–	–	–	–	–	–	–	–	–	–	–	–
12	Hopping alien #3(M)	33	10	0	8	0	–	–	–	–	–	–	–	–	–	–	–	–
13	Hopping alien #4(M)	33	10	0	8	0	–	–	–	–	–	–	–	–	–	–	–	–
14	Square	16	4	0	0	5	–	–	–	–	–	–	–	–	–	–	–	–
15	Stone call section	24	10	0	0	9	–	–	–	–	–	–	–	–	–	–	–	–
16	Ladder section	24	10	19	0	0	–	–	–	–	–	–	–	–	–	–	–	–
17	Data sprite	8	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
18	Data sprite	8	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
19	Data sprite	10	18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
20	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
21	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
22	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
23	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
24	Chasing alien #1(M)	33	14	0	0	14	–	–	–	–	–	–	–	–	–	–	–	–
25	Chasing alien #2(M)	33	14	0	0	14	–	–	–	–	–	–	–	–	–	–	–	–
26	Chasing alien #3(M)	33	14	0	0	14	–	–	–	–	–	–	–	–	–	–	–	–
27	Chasing alien #4(M)	33	14	0	0	14	–	–	–	–	–	–	–	–	–	–	–	–
28	Long cube	15	5	0	3	6	–	–	–	–	–	–	–	–	–	–	–	–
29	Data sprite	3	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
30	Eyeball #1	16	6	0	16	9	–	–	–	–	–	–	–	–	–	–	–	–
31	Eyeball #1	16	6	0	16	9	–	–	–	–	–	–	–	–	–	–	–	–
32	Eyeball #1	16	6	0	16	9	–	–	–	–	–	–	–	–	–	–	–	–
33	Eyeball #1	16	6	0	16	9	–	–	–	–	–	–	–	–	–	–	–	–
34	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
35	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
36	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
37	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
38	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
39	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
40	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
41	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
42	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
43	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
44	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
45	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
46	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
47	Data sprite	1	4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
49	Small ladder section	8	3	0	11	0	–	–	–	–	–	–	–	–	–	–	–	–
50	Corkscrew #1	20	4	0	7	3	–	–	–	–	–	–	–	–	–	–	–	–
51	Corkscrew #2	20	4	0	7	3	–	–	–	–	–	–	–	–	–	–	–	–
52	Corkscrew #3	20	4	0	7	3	–	–	–	–	–	–	–	–	–	–	–	–
53	Corkscrew	20	4	0	7	3	–	–	–	–	–	–	–	–	–	–	–	–
55	Animated toilet #1	19	7	14	23	24	–	–	–	–	–	–	–	–	–	–	–	–
56	Animated toilet #2	19	7	14	23	24	–	–	–	–	–	–	–	–	–	–	–	–
57	Animated toilet #3	19	7	14	23	24	–	–	–	–	–	–	–	–	–	–	–	–
58	Animated toilet #4	19	7	14	23	24	–	–	–	–	–	–	–	–	–	–	–	–
63	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
64	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
65	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
66	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
67	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
68	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
69	Data sprite	8	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

APPENDIX C(ii) — SPT4 SPRITES (sprites used in the demo part 2)

```
To load sprites into Laser BASIC, type: A$="SPT4SPR":|GSPR,@A$
```

Sprite No.	Description	Suggested alternative colours for the inks																
		HGT	LEN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	OCEAN IQ Logo	33	28	9	21	23	-	-	-	-	-	-	-	-	-	-	-	-
70	Stone brick	16	8	0	0	13	12	0	0	0	0	0	0	0	0	0	0	0
71	Stone brick with weeds	16	8	0	24	13	12	1	0	0	0	0	0	0	0	0	0	0
72	Walking monk #1(M)	26	10	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
73	Walking monk #2(M)	26	10	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
74	Walking monk #3(M)	26	12	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
75	Climbing monk #1(M)	27	12	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
76	Climbing monk #2(M)	27	10	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
77	Climbing monk #3(M)	26	12	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
78	Stone window	36	7	0	0	12	25	0	0	0	0	0	0	0	0	0	0	0
79	Spearman #1	38	5	0	0	15	3	12	9	0	0	19	16	0	0	0	0	0
80	Spearman #2	38	5	0	0	15	3	12	9	0	0	19	16	0	0	0	0	0
81	Spearman #3	40	5	0	0	15	3	12	9	0	0	19	16	0	0	0	0	0
82	Pointing monk	29	5	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
83	Monk's hand	6	5	0	0	0	0	0	12	25	15	0	13	0	0	0	0	0
84	Woman	35	3	0	11	0	0	2	1	26	15	21	0	0	0	0	0	0
96	Rocket plane #1(M)	14	24	0	0	0	0	14	24	25	0	16	0	0	0	0	0	0
97	Rocket plane #2(M)	14	24	0	0	0	0	14	24	25	0	16	0	0	0	0	0	0
98	"Press any key" (M)	8	50	14	0	-	-	-	-	-	-	-	-	-	-	-	-	-
100	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
101	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
102	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
103	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
104	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
105	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
106	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
107	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
108	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
109	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
110	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
111	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
112	Data sprite	1	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

APPENDIX C(iii) — SPT5 SPRITES (sprites used in the demo part 3)

```
To load sprites into Laser BASIC, type: A$="SPT5SPR":|GSPR,@A$
```

Sprite No.	Description	Suggested alternative colours for the inks																
		HGT	LEN	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
60	Platform game Channel A music sprite	7	49	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
61	Platform game Channel B music sprite	1	128	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
62	Platform game Channel C music sprite	1	152	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
86	Loading screen Channel A music sprite	2	234	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
87	Loading screen Channel B music sprite	1	94	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
88	Loading screen Channel C music sprite	1	94	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
90	Ocean IQ screen Channel A music sprite	3	119	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
91	Ocean IQ screen Channel B music sprite	2	128	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
92	Ocean IQ Channel C music sprite	1	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
93	Hunch back screen Channel A music sprite	3	91	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
94	Hunch back screen Channel B music sprite	1	237	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
95	Hunch back screen Channel C music sprite	2	122	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

NOTES

NOTES

NOTES

TECHNICAL ENQUIRY CARD

WE AT OASIS BELIEVE IN GIVING FULL TECHNICAL SUPPORT TO ALL OUR PRODUCTS. TO ASSIST US, PLEASE FILL IN THIS CARD AND RETURN TO THE ADDRESS BELOW. TELEPHONE ENQUIRIES, OR LETTERS NOT ACCOMPANIED BY THIS CARD CANNOT BE ANSWERED. A REPLACEMENT CARD WILL BE ENCLOSED WITH OUR REPLY. ALL ENQUIRIES WILL BE ANSWERED AT OUR FIRST OPPORTUNITY.

MAKE AND MODEL OF COMPUTER.

SOFTWARE PURCHASED.

VERSION No.

PLACE AND DATE OF PURCHASE.

YOUR NAME.

ADDRESS.

TELEPHONE No.

AGE OCCUPATION.

PLEASE WRITE YOUR ENQUIRY HERE.

Please return to:-
OASIS SOFTWARE, 12 WALLISCOTE ROAD, WESTON-SUPER-MARE,
AVON, ENGLAND. BS23 1UG.