WHITE LIGHTNING

by OASIS SOFTWARE

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 the 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.

CONTENTS

WHITE LIGHTNING
by Oasis Software

INTRODUCTION

White Lightning is a high level graphics development system for the Spectrum 48k. It is aimed primarily at the user who has commercial games writing in mind and has the patience to learn a sizeable new language. It is not a games designer and stunning results probably won't be produced overnight, but it does have the power and flexibility to produce software of a commercial standard (with a little perseverance!). Software produced using White Lightning can be marketed without restriction, although, we would be very grateful if you felt you could pop a small credit on the sleeve. If you're looking for a publisher - don't forget us!

Assembly language has three advantages over high level languages: speed, flexibility and compactness. During the running of an arcade game, the processor spends most of its time manipulating screen data, and if the appropriate commands are implemented in the language, the execution "overhead" is very small. Add to this the fact that considerable time has been spent on the routines themselves to optimise execution speed, and we feel most machine code programmers would be hard pressed to better White Lightning for speed. As far as flexibility is concerned, White Lightning has almost 300 commands as well as access to BASIC and machine language if required. A lot of the tricky routines like rotations and enlargements are already implemented for you. As far as compactness goes, Forth itself produces very compact code, but there is, of course, the overhead of the language itself. Assembly language has four major drawbacks. Firstly, you've got to learn it. Having mastered machine code, program development is very slow compared with a typical high level language, there is no "crash protection" whatsoever, and to produce effective results, you need a fairly intimate knowledge of the machine you're working with.

BASIC has several points in its favour, these are: excellent crash protection, extremely readable source code and a relatively short learning curve. These features make BASIC a very good introduction to programming for the hobbyist, but for the serious games writer, the language is insufficient in terms of both its speed and flexibility.

Because White Lightning is Forth based it has virtually the speed of machine code, no knowledge of the machine is required, the source code is relatively readable, and it is fairly well protected from crashing.

If you do have any queries concerning White Lightning, then we can be contacted by phone on (0934) 419921. If possible, please restrict calls to the periods 9 am to 11 am or 6 pm to 6.30 pm. If this is not convenient we are here all day. If your query is a detailed one then it's probably better to write in. We are also interested to hear of any extensions or routines you may develop, and if sufficient interest is shown we will start a News Letter, and possibly, even a User Group.

SPRITE DEVELOPMENT

Included with White Lightning is a sprite generator. This comes complete with a pre-defined character set which, when suitably combined, makes up to 167 full characters. The predefined characters cover Asteroids, Pac-Man, Scramble, Defender, Space Invaders, City Bomber, Lunar Lander, Frogger, Centipede, Donkey Kong and many more. You can use them as they are, customise them, or design up to 255 of your own sprites. The development software allows you to reflect, spin or invert. When you have finished work, or between sessions the whole lot can be simply saved to tape.

IDEAL

The main part of the package is the White Lightning language itself. The language can be thought of as being divided into two parts: firstly, there is a super fast integer Forth, which conforms to a standard Fig-Forth, but secondly, and of most importance to games designers, there is the IDEAL sub-language. IDEAL stands for "Interrupt Driven Extendable Animation Language". IDEAL has a dictionary of over 100 words, which can be freely mixed with Forth, or, as we shall see, can be accessed from BASIC.

Interrupt Driven

Forth/IDEAL words can be executed under interrupt; this means that programs can be run in foreground and background at the same time. Suppose, for instance, the program you are writing involves a scrolling backdrop, which has been defined in a sprite 6 screens wide. A program can be run in background to handle the scrolling backdrop, and a separate program written in foreground to control all of the characters which move within the backdrop. This will free the user from complex timing calculations to get a smooth scroll and is one of the most powerful features of the entire package. Background words can be executed up to 50 times a second.

Extendable

Forth is extendable and was chosen as the most suitable host language for IDEAL because of this extremely useful feature. New words can be defined in terms of any of the Forth/IDEAL words, or your own previously defined words. This means you can create diagonal scrolls for instance, by combining individual scrolls.

Animation Language

Very careful planning went into the designing of the IDEAL animation language, to make it as portable as possible between micros. If you've written a very successful program for one micro, you can move the same code across without too much difficulty. The only major changes necessary, are changes to accommodate the different screen formats, ie the number of columns and rows. You don't need to worry about the complexities of various memory mappings, IDEAL does that for you. Some implementations will not allow attributes to be moved separately from pixel data, so to be certain of higher portability, move attributes and data together.

ACCESS FROM BASIC

If you are not familiar with Forth and want to get reasonable software quickly, you can access the IDEAL language from BASIC. Programs will no longer be portable and you won't get quite the same speed and polish, but perfectly good programs can, and have, been written this way. More memory will be used for BASIC source, so bear all this in mind before deciding to put off learning Forth! Most of the useful BASIC commands that handle sound and graphics, such as DRAW, CIRCLE, BEEP and so on, have been implemented in Spectra Forth. Whenever possible, call them from Forth and not BASIC. The interpretation and initial floating point manipulation of BASIC commands are avoided, and commands will therefore execute more rapidly from Forth.

OPERATING INSTRUCTIONS

1) Rewind the tape marked "White Lightning". Disconnect Interface 1 if fitted.

2) Load using LOAD"" then stop the tape. White Lightning will auto-run.

3) Once loaded, you will get the prompt LOAD SPRITES Y/N.

4) If it's the first time you've used the package, type Y to load the demonstration sprites, which will follow directly after White Lightning on the same tape. There is a section of data before the sprites which doesn't load, but don't worry about this, this is information used by the sprite development package. Once loaded, the LOAD SOURCE Y/N prompt will appear. Press N to enter White Lightning or, if you wish to load some source code place the appropriate cassette in the recorder, press PLAY and then type Y.

5) To RUN the demonstrations, just LOAD using LOAD"" and the program will auto-run.

6) To RUN the Sprite Development Software just type LOAD"" and again the program will auto-run.

TO THE NEWCOMER

When you have run the demonstration tape and have seen what can be done, this may give you the incentive to learn all you can to produce full specification games for yourself. The author of this manual knew nothing of White Lightning before starting this project, but can promise you that after only a few hours of experimentation, became fairly adept and had the confidence to want to go further. It may all look a little complicated at first, but please be assured, that after a short time, and only a little effort, the fog really does clear!

Mike Butler.

SECTION 1

THE SPRITE GENERATOR PROGRAM
by Paul Newnham

INTRODUCTION

The Sprite Generator Program was developed to complement the White Lightning language. The language is comprised of commands for manipulating sprites and screen data but does not have the facility to directly design graphics characters. This means there are two phases to games creation. The first involves designing and editing your graphics characters with the sprite generator program, and the second involves the writing of the game itself using the White Lightning language. In practice the two areas of work will probably be carried out simultaneously. For those of you who are not artistically inclined, there are two sets of previously defined graphics characters ready to use.

The Arcade Character Set

The arcade character set is an integral part of the sprite generator program - 167 characters are provided in all. To see these, LOAD and run DEMO B using LOAD "". The demo will auto-run. This will tell you which characters can be called up by which number, using Function Key Z. These characters are summarised as the penultimate part of this section.

The Demonstration Sprites

Directly after White Lightning on side A you will find the demonstration sprites. By running the tape past White Lightning and using the LOAD SPRITES FROM TAPE facility, these can be loaded and edited for your own use. The various sprites are tabulated at the end of this section.

USING SPRITES WITH WHITE LIGHTNING

Once you have completed an editing session, the sprites generated should be saved to tape for further editing sessions, or for use with the White Lightning language itself. To load your sprites into White Lightning:

1. Load White Lightning using LOAD ""

2. Insert the tape containing your sprites into the tape recorder and press Y in response to the "LOAD SPRITES Y/N" prompt. The sprites will be loaded at the address from which they were saved using the sprite generator program.

COLD START

If you enter the sprite generator program via a COLD start, then all sprites previously stored will be cleared and all system variables reset. If, for instance, you wish to use the demonstration sprites, you would enter via a COLD start. The program must always be initially entered via a COLD start.

WARM START

If you enter the program via a WARM start then all sprites will be conserved and all system variables left unchanged. It is provided principally for re-entering the program after an accidental BREAK or ERROR. If you do accidentally BREAK; type: GOTO 3 and then enter via the WARM start.

BUFFER SIZE

When White Lightning runs programs in Background mode (see Section 3) the top end of memory is used as a scratch pad. The size of this area depends on the operation of the program and calculating the amount you need to reserve is covered in Section 3. When the sprite generator program is entered the buffer has a default size of 256 bytes. This is probably much larger than required, but until you are familiar with the package or need to save a few extra bytes, just leave the buffer at 256 bytes.

THE CHR$ SQR

CHR$ SQR is the abbreviation used throughout this text for character square, and refers to the 8 by 8 grid to the left of the sprite screen. This is the area used to create and edit sprites one character at a time.

THE SPRITE SCREEN

This is the area of screen 15 characters by 15 characters on which sprites are created, developed, transformed and generally worked on.

THE CHR$ SQR CURSOR

This is the non-destructive flashing cursor which is used to design and edit the character currently held in the CHR$ SQR.

THE SPRITE SCREEN CURSORS

These are the two flashing cursors displayed in the row beneath the sprite screen and the column to the right of the sprite screen. They are used to indicate the position of the top left hand corner of the screen window currently being operated upon. The actual cursor positions are measured from the top left hand corner of the sprite screen and are displayed in real time on the screen as X POS (column) and Y POS (row). Top left is X POS 1 Y POS 1. Bottom right is X POS F Y POS F.

SCREEN WINDOWS

The area of the screen currently being worked on is referred to as the screen window. Its position is defined by X POS and Y POS, which correspond to the positions of the sprite screen cursors, and its dimensions are defined by SPRITE HEIGHT and SPRITE LENGTH. To see the screen window you are currently working on just press F. The window will flash.

SPRITE LIBRARY

This refers to the set of sprites you are currently working with and can contain up to 255 sprites or use 12500 bytes. If your sprite library needs more than 12500 bytes you can use the merging procedure detailed in section 3 to load and merge more than one sprite library into White Lightning.

OPERATING INSTRUCTIONS

Insert the Sprite Generator Program Tape, type LOAD"" and load into the computer as normal.

Once loaded, the program will auto-run and the screen message "COLD OR WARM START" will appear. If this is the first execution of the program or if you wish to clear the sprite memory, press C for a COLD START.

A further screen message will now appear asking if you wish to change the buffer size - this has a default value of 256 bytes. For now, press N. This function will become more apparent later.

NOTES:

1. A WARM START will not destroy any sprites already in memory and if ever the program is accidentally caused to BREAK, type GOTO 3 and in response to the screen prompt, execute a WARM START.

2. A COLD START will destroy any and all sprites defined in memory.

3. WARM STARTs can only be executed after an initial COLD START.

GETTING FAMILIAR WITH THE FUNCTION KEYS

Now let's get familiar with some of the Function Keys - a full list will be found at the end of this section.

THE CHR$ SQR

This is the grid square on which you create and edit characters for your own sprite library. To move the cursor:

1. Press the 5 key for each movement to the left.

2. Press the 6 key for each movement downward.

3. Press the 7 key for each movement upward.

4. Press the 8 key for each movement to the right.

Now that you know how to move the cursor, let's fill in a few squares:

1. Move the cursor to any square that you like and release the keys.

2. Press the 9 key to set the square.

3. Now move the direction keys and fill in a few more squares.

Now that we have set some squares, what about deleting a few of then? This is simple:

1. Move the cursor to a square that you have set and release the keys.

2. Press the 0 key to clear the square.

Now have a go at setting and clearing some squares, just to get used to it.

The Sprite Screen Cursors

Now that you're used to moving the CHR$ SQR cursor around, moving the sprite screen cursors is a piece of cake:

1. Move the X cursor by pressing SYMBOL SHIFT and the 5 or 8 key to move left or right respectively.

NOTE: If you've never been able to remember which is the X or which is the Y movement, remember this little saying:
X is a cross - if you say it quickly it sounds like, X is across - which it is!

2. Move the Y cursor by pressing SYMBOL SHIFT and the 7 or 6 key to move up or down respectively.

Character Building

No, not yours - building up characters to make up sprites! You've probably got quite a mess in the CHR$ SQR, so let's clear it:

1. Press the Q key and respond to the prompt in the text line by pressing Y and the CHR$ SQR will clear.

Just to get you used to a similar function, let's clear the Sprite Screen as well, even though it's clear:

1. As you can see, to clear the CHR$ SQR press Q, to clear the sprite screen press SYMBOL SHIFT Q - cunning, eh?

Now that we have clear screens we can start to go places. Have a go at this:

1. Move the X and Y cursors to 1 and 1 respectively.

2. Press the Z key (to call up a character from the Arcade Library) and enter the number 63 followed by ENTER. A space invader type character will appear on the sprite screen.

3. Press the K key and answer Y to the prompt and hey presto - the character has been placed in the CHR$ SQR.

This will illustrate quite nicely how a character is built up.

Have a go at changing this character using the 5, 6, 7 or 8 keys to move the CHR$ SQR cursor and the 9 and 0 keys to set or clear a square. You won't be able to see the CHR$ SQR cursor at the moment - just press one of the cursor keys and it will flash for you.

Let's GET your 'new character' into memory:

1. Move the sprite screen cursors to X POS 4 and Y POS 4 (SYMBOL SHIFT 5, 6, 7, and 8)

2. Press the J key and answer Y to the question.

3. Your new character was placed on the sprite screen from the CHR$ SQR by using the J key. Your original character is still there at X POS 1 Y POS 1.

4. Now press the S key to give your character a sprite number. For now, just enter the number 1 and press ENTER.

5. Press the G key to GET the character into memory as a sprite and answer Y to the prompt - your character will flash to confirm.

Now, let's prove that your character is in memory:

1. Press the SYMBOL SHIFT Q to clear the sprite screen.

2. Press S to tell the computer which sprite you are calling up (there is only one at the moment of course). Enter 1 followed by ENTER.

3. Now press P to PUT the sprite to the sprite screen and answer Y to the question.

4. Now you will be given four more options. Don't worry about 2, 3 and 4 for now - we just want to place our sprite on the sprite screen. Press 1, and there it is!

You will notice that although you correctly got your own sprite PUT back to the sprite screen, the original character from the Arcade Library wasn't. This example was to show you that any actions that you call for, will only happen to the character that the sprite screen cursors are pointing to, as we pointed out in the Introduction.

You will also notice that the CHR$ SQR still contains your new character - have a look to compare.

We have seen how to call up a character from the Arcade Library and how, in essence, to build up a character in the CHR$ SQR. There is another way to build up a character:

1. Press SYMBOL SHIFT Q to clear the sprite screen.

2. Press the D key, answer Y to the question, and enter the following, very carefully, pressing ENTER after each entry:

    a) H24 126 H9D 255 HFF 153 129 102

3. Guess who's back! (You should have a space invader type character).

This is the DIRECT DATA INPUT. Direct Data characters are built up from 8 bytes of data, one byte at a time.

NOTE: Data can only be entered using values in the range 0 to 255 Decimal or H00 to HFF HEX. The character H must precede a HEX entry.

Let's do a quick review of the functions that we have used:

1. CHR$ SQR - cursors, 5, 6, 7 and 8 keys to move and the 9 and 0 keys to set and clear squares. The Q key clears the CHR$ SQR.

2. SPRITE SCREEN - 5 and 8 to move the X cursor (X is across remember) and 6 and 7 to move the Y cursor. The SYMBOL SHIFT and Q keys clear the sprite screen.

3. The Z key calls up the ARCADE CHARACTER LIBRARY - 1 to 167.

4. The K key transfers a character from the SPRITE SCREEN to the CHR$ SQR.

5. The J key transfers a character from the CHR$ SQR to the SPRITE SCREEN.

6. The S key defines and subsequently calls up a particular SPRITE.

7. The G key GETs your sprite into memory.

8. The P key PUTs your sprite from memory onto the SPRITE SCREEN.

9. The D key enables you to enter a character by DIRECT DATA to the SPRITE SCREEN.

You've used quite a few functions! Have a go at calling up some more Arcade Characters, change them if you wish, then GET them into memory and PUT them onto the SPRITE SCREEN.

The Information Rectangle

```
MEMORY LEFT 12486  X POS 4  Y POS 4
SPRITE 65266   SPRITE HEIGHT-1
SPST.. 65266   SPRITE LENGTH-1
SPND.. 65280   SPRITE NUMBER-1
     The text line
```

This is a most useful facility which can be of great service to you. Most of the information is fairly obvious, but we'll run through it all:

1. MEMORY LEFT, as it says, is the amount of memory available for sprites; these are the sprites that you define and do not include the Arcade Characters - to use these, of course, you must define them as sprites by GETting them into memory.

2. X POS Y POS, these are the current positions of your SPRITE SCREEN X and Y cursors with reference to the figures on top and to the left of the SPRITE SCREEN.

3. SPRITE, this indicates the position, in memory, where your defined sprite is.

4. SPST, indicates the SPrite space STart point, in memory. (Before any sprites are defined this has an initial value of 65280).

5. SPND, indicates the SPrite space eND point, in memory.

6. SPRITE HEIGHT, indicates the height of your defined sprite, in character squares, as indicated by the figures at the top and to the left of the SPRITE SCREEN. (This has an initial value of 1).

7. SPRITE LENGTH, indicates the length of your defined sprite, in character squares, as indicated by the figures at the top and to the left of the SPRITE SCREEN. (This has an initial value of 1).

8. SPRITE NUMBER, indicates the sprite currently defined. (This has an initial value of 1).

9. The Text Line, to show the Function called up, and the available options.

More Function Keys

Let's move on. BREAK the program - CAPS SHIFT/BREAK keys, type GOTO 3 and press ENTER. Execute a COLD (C) START and answer N to BUFFER SIZE CHANGE. Nothing of what you have previously done is in memory. (We could have cleared all your sprites by defining them (S key) and pressing W (WIPE SPRITE), but depending on how many you defined whilst experimenting, it could have been a lengthy process!)

Now have a go at this:

1. Clear the sprite screen (SYMBOL SHIFT Q)

2. Press X to activate the INK variable and then set it to 2.

3. Press C to activate the PAPER variable and then set it to 7.

4. Press B to activate the BRIGHT switch and then press 1 to switch it ON.

5. Press V to activate the FLASH variable and then press 0 to switch it OFF.

6. Press A to activate the ATTRIBUTE switch and then press 1 to switch it ON.

You will have noticed, that both PAPER and FLASH were already set to 7 and 0 respectively from the COLD start; we only run through them all for completeness and to get used to using them.

What you have done, is to set the attributes for the character we are about to define, so lets do that:

1. Press Z to call up the arcade characters, answer Y to the prompt, and enter 150 followed by ENTER - there you have it - a red Dalek.

Now in order to define this character as a sprite, we reed to GET it into memory.

1. Press S to set up a sprite and type 10, followed by ENTER.

Now we'll need to set up the screen window:

1. Press L to activate the sprite length variable and then press 2 followed by ENTER. You will see the window flash red across the top half of the character.

2. Press H to activate the SPRITE HEIGHT variable and then press 2 followed by ENTER. Now you will see the new window flash.

Now convert this character into a sprite by GETting it into memory:

1. Press G to activate the GET function and answer Y to the prompt. Again the screen window will flash, confirming that the character has been GOT into memory.

Move the sprite screen X cursor +2 (SYMBOL SHIFT 8 twice). Activate the PUT function (P), respond with a Y to the prompt and press 1 to PUT your new sprite to the screen.

Now we're going to mirror this second character and GET it into memory as another sprite but with different attributes:

1. Press E (Screen Functions), answer Y and press 2 (Mirror) - the character is reflected!

2. Now press S (Sprite Number) and enter 11.

3. Press G (GET) and then Y - again it flashes to confirm.

We now have two sprites, one facing left and one facing right. Let's set some new attributes.

1. Press X (INK) and enter 4 (green).

2. Press I to activate the Attribute Dump facility.

3. Move the sprite screen X cursor 1 place to the right and press I again - the top half is done!

4. Move the sprite screen Y cursor 1 place down and press I.

5. Finally, move the sprite screen cursor one place to the left and press I again - there you have it, a red character to the left and a green one to the right.

If you want to swap the colours the other way around - yes you're right this can be done:

1. Position the sprite screen cursors to X POS 1 Y POS 1.

2. We will have to set up the screen window for the exchange:

Press L (Length Variable) and enter 4 - the window will flash.

3. Press E (Screen Functions), answer Y to the prompt and press 3 to MIRROR ATIRIBUTES - watch the screen as you press 3, it happens very quickly! The red Dalek becomes green and the green Dalek becomes red.

Right then - we'll move on a little. We'll consider some sprite operations. These are operations which take place in memory on the stored sprites. We'll begin by setting up a new sprite comprising two arcade characters and then go on to change their positions in the sprite in memory. Have a go at this:

1. Clear the sprite screen (SYMBOL SHIFT Q). Don't worry about your previous characters - you'll probably remember that they are still in memory as sprites 10 and ll.

2. Make sure that the sprite screen cursors are set to X POS 1 Y POS 1.

3. Set INK (X) to 6 (yellow).

4. Set PAPER (C) to 0.

5. Set FLASH (V) to 0.

6. Set BRIGHT (B) to 1.

7. Set ATTR (A) to 1.

8. Press Z (Arcade Character) and enter 149 (a Robot).

9. Move the sprite screen X cursor by +2.

10. Change the INK (X) to 4 (green).

11. Press Z (Arcade Character) and enter 151 (another Robot).

OK, you should have two robots on the sprite screen - let's define them as a combined sprite:

1. Move the cursors to X POS 1 Y POS 1.

2. Press L (Length Variable) and set to 4.

3. Press H (Height Variable) and set to 2.

4. Press S (Sprite Number) and enter 12.

5. Press G (GET Function) and respond to the prompt with Y.

We have now set up a 4 by 2 sprite containing both characters. Now let's change them over in memory:

1. Press M (Sprite Memory Functions), respond to the prompt with Y, and press 2 (MIRROR) - nothing happens on the sprite screen - this is a memory function.

Let's prove that the sprite has been altered:

1. Move the sprite screen Y cursor by +3.

2. Press P (PUT) respond to the prompt with Y and press 1 - the sprite has been reflected.

We can return the attributes to their former Robots quite easily:

1. Press M (Sprite Memory Functions), respond to the prompt with Y, and press 3 (Mirror Attributes).

2. Move the Y cursor by +3 and press P (PUT), respond to the prompt with Y and press 1 - now the attributes have mirrored.

Let's move on a little further now. We now look at a second sprite transformation - rotation. Try the following:

1. Clear the sprite screen and set the X and Y cursors to X POS 1 Y POS 1.

2. Set INK (X) to 1 (blue).

3. Set PAPER (C) to 7 (white).

4. Set FLASH (V) to 0.

5. Set BRIGHT (B) to 1.

6. Set ATTR (A) to 1.

7. Press Z (Arcade Character) and enter 151 (a Robot).

8. Set HEIGHT (H) to 2 and LENGTH (L) to 2.

9. Set the Sprite Number (S) to 13, then press G to GET the character as a sprite.

10. Move the X cursor by +2.

11. Press R (ROTATE), and enter the new sprite number, 14.

12. Press P (PUT) and press 1 - a Robot rotated by 90 degrees!

Now that we have sprite 14 as a 90 degree rotation of sprite 13, why not go a little further ? Try this:

1. Move the X cursor by +2.

2. Press R (ROTATE) and enter the new sprite number, 15.

3. Press P (PUT) and press 1 - this new robot has been rotated by 180 degrees from its original orientation.

To produce the final (270 degree) orientation:

1. Move the X cursor by +2.

2. Press R (ROTATE) and enter the new sprite number, 16.

3. Press P (PUT) and press 1 - this has produced the final orientation.

Now let's look at attribute handling in more detail - clear the sprite screen and position the X and Y cursors to X POS 1 Y POS 1. The following two examples will show how to download and pick-up attributes between the attribute variables and the sprite screen:

1. Press X (INK) and set to 3 (magenta).

2. Press C (PAPER) and set to 2 (red).

3. Press V (FLASH) and set to 1 (ON).

4. Press B (BRIGHT) and set to 0 (OFF).

5. Press A (ATTR) and set to 0.

6. Press I (ATTRIBUTE DUMP) - the attributes will appear on the sprite screen.

7. Now set all the attributes, X, C, V, B, and A to 0.

8. Press U (PICK UP ATTRIBUTES) and the attributes on the screen will be loaded into the attribute variables.

This next example illustrates one of the more complicated functions of the generator - GETting a sprite into a larger sprite:

1. Clear the sprite screen and position the cursors to X POS 1 Y POS 1.

2. Press L (Length) and enter 4, then press H (Height) and enter 4.

3. Press E (Screen Functions) and press 1 (INVERT).

4. Press S (Sprite Number) and enter 17, then press G (GET).

5. Move the X cursor by +4.

6. Press X (INK) and set to 0.

7. Press C (PAPER) and set to 6 (yellow).

8. Press V (FLASH) and set to 0 (OFF).

9. Press B (BRIGHT) and set to 1 (ON).

10. Press A (ATTR) and set to 1.

11. Press Z (Arcade Character) and then enter 149.

12. Press L (Length) and enter 2, then press H (Height) and enter 2.

13. Press S (Sprite Number) and enter 18.

14. Press G (GET).

15. Press SPACE key (Place small sprite into large sprite) - enter small sprite as 18, larger sprite as 17, Row as 1, Column as 1 and BLS (1), as you get each prompt.

16. Move the X cursor by +2.

17. Press P (PUT), answer the prompt with Y and then press 1 - there you have it, your Robot inside the square.

Let's move on to look at some of the utility functions - Test and Wipe.

1. Clear the sprite screen and position the cursors to X POS 1 Y POS 1.

2. Press Z (Arcade Character) and enter 135.

3. Press L (Length) and enter 2, then press H (Height) and enter 2.

4. Press S (Sprite Number) and enter 19.

5. Press G (GET).

6. Press L (Length) and enter 10, then press H (Height) and enter 10.

7. Clear the sprite screen.

8. Press T (Test Sprite) and note that the information rectangle contains the following:

```
A)  SPRITE HEIGHT = 2
B)  SPRITE LENGTH = 2
C)  SPRITE NUMBER = 19
D)  MEMORY LEFT   = The remaining memory for sprites.
E)  SPRITE        = Start address of sprite being tested.
```

A function is provided to Wipe a sprite from memory and adjust pointers. Leave everything as it is and try:

1. Press W (Wipe sprite) - remember sprite 19 has already been defined. Respond to the prompt with Y.

2. Press T (Test sprite) and an error message will appear on the text line - SPRITE NO LONGER EXISTS

While we are considering error messages, have a go at this:

1. Press Z (Arcade Character) and then press 63.

2. Press S (Sprite Number) and press 12 - an error message will appear, SPRITE ALREADY DEFINED. Sprite number 12 is not corrupted in any way, nor is the one you have put to the screen. All that you need to do is choose a different sprite number which has not already been allocated.

There's one area we've been avoiding all the way through - the Logic Functions. We don't want to wade into the depths of Boolean algebra here, but instead, provide a few examples which hopefully show the application of the XOR, OR and AND operations to this part of the package. They are provided for advanced applications only and their results are summarised on the sprite generator panel. Let's see what they do:

1. Clear the sprite screen and position the cursors to X POS 1 Y POS 1.

2. Press X (INK) and set to 3 (magenta).

3. Press C (PAPPR) and set to 0 (black).

4. Press V (FLASH) and set to 0.

5. Press B (BRIGHT) and set to 1.

6. Press A (ATTR) and set to 1.

7. Press Z (Arcade Character) and enter 149 (Robot).

8. Press L (Length) and enter 2, then press H (Height) and enter 2.

9. Press S (Sprite Number) and enter 21, then G (GET) to Get the sprite.

10. Move the X cursor by +2.

11. Press P (PUT) and respond to the prompt with 1.

12. Press Z (Arcade Character) and enter 151 (Robot).

13. Press O (Logical Sprite Functions) and press 1.

14. Move the X cursor by +2.

15. Press P (PUT) and respond with 1 - the characters have been "ORed" (merged).

Let's take this a little further and OR with the screen:

1. Clear the sprite screen and position the cursors to X POS 1 Y POS 1.

2. Press X (INK) and set to 1 (blue).

3. Press C (PAPER) and set to 7 (white).

4. Press V (FLASH) and set to 0.

5. Press B (BRIGHT) and set to 1.

6. Press A (ATTR) and set to 1.

7. Press Z (Arcade Character) and enter 141 (Explosion) - you probably think we're going to blow up the Robot!

8. Press L (Length) and enter 2, then press H (Height) and enter 2.

9. Press S (Sprite Number) and enter 22, then G (GET) sprite 22.

10. Move the X cursor by +2.

11. Press Z (Arcade Character) and enter 149 (Robot).

12. Press P (PUT) and enter 2 - sprite number 22 has been "ORed" with whatever was on the screen.

13. Move the X cursor by +2.

14. Press P (PUT) and enter 1 - the explosion was unaffected!

Now let's take a look at a second logical operation - the AND function:

1. Clear the sprite screen and position the cursors to X POS 1 Y POS 1.

2. Press P (PUT) and enter 4, sprite 22 has been "ANDed" with the screen - as the screen was empty, nothing happened.

3. Press P (PUT) and enter 1 - the sprite is still there!

Now, lastly, let's look at the XOR function:

1. Clear the sprite screen and move the cursors to X POS 1 Y POS 1.

2. Press Z (Arcade Character) and enter 151 (Robot).

3. Press P (PUT) and enter 3 - the Robot has blown up! ("At last!", I hear you say!)

Now one of the interesting properties of the logical XOR, is that if the operation is repeated, the original character is restored - to mend the Robot:

1. Press P (PUT) and enter 3 - a fully restored Robot!

Well, there you have it. In fact, the only functions remaining that have not been covered by an example are:

1. Relocate Sprites (< (SYMBOL SHIFT R)).

2. Save sprites to Tape (NOT (SYMBOL SHIFT S)).

3. Load Sprites from Tape (- (SYMBOL SHIFT J)).

The "Relocate Sprites" does simply that, it relocates sprites in memory. Just enter a relocation length - positive to move sprites to higher memory and negative to move sprites to lower memory. This function is not often used in fact and should be used with great care!

The "Save Sprites" function is very straight forward - just follow the screen instructions. After the sprites have been saved you will be asked to rewind the tape to verify the saved data.

The "Load Sprites" function is similar to the above - just follow the screen instructions.

It should be noted that whenever sprites are saved or loaded, there are 3 distinct sets of data which the generator stores and retrieves.

CONCLUSIONS

Well, there's nothing to stop you now! Going through the examples will help you to familiarise yourself with all of the Function Keys. It won't be long before you'll be happy to press on by yourself - happy sprite generating!

The thing to do now, is to press on with Spectra Forth and IDEAL, these really are not as frightening as they may first appear - after seeing the demo program, what more incentive do you need!

SECTION 2

SPECTRA FORTH
by Stuart Smith

Forth is an extraordinary computer language developed originally for the control of radio telescopes, by an American named Charles Moore.

Forth is neither an interpreter nor a compiler, but combines the best features of both to produce a super-fast, high level language, incorporating the facilities offered by an interactive interpreter and the speed of execution close to that of machine-code. In order to achieve these fantastic speeds, Forth employs the use of a data, or computation stack, on which to hold the data or the operations to be performed, coupled with the use of Reverse Polish Notation (RPN). This may be quite a mouthful, but RPN is very easy to use and understand with only a little practice - in fact, Hewlett Packard use RPN on many of their calculators.

All standard Forths use integer arithmetic for their operations and can handle up to 32 bit precision if required - floating point mathematics routines could be incorporated, but with a reduction in the execution speeds of a program.

White Lightning consists of a standard Fig-Forth model, but with over 100 extensions to the standard vocabulary of Forth words. There are two important extensions to White Lightning: the first is the ability to access almost ALL of the Spectrum's own BASIC commands, just as you would when writing a BASIC program, and with the addition of many of the high resolution graphics commands (CIRCLE, DRAW, etc.). Coupled with the incredible execution speeds of White Lightning, the possibilities are limitless! The second, and possibly most important, addition is the IDEAL sub-language.

In addition to the basic vocabulary of White Lightning words, the user can very easily ADD his own NEW WORDS using previously defined words, thus extending the vocabulary and building up as complex a word as is necessary to do the task in hand.

Fully structured programming methods are also employed as a fundamental feature of Forth through the use of the structured control sequences included, such as:

```
IF.....ELSE.....ENDIF
DO.....UNTIL
```

The standard Spectrum editor can be used to create lines of White Lightning source code for later compilation. Do not allow lines to exceed 64 characters - any characters after this are ignored. The standard Forth line editor is included for compatibility with existing text. The source code is stored in memory from $CC00 onwards, and can be LOADed or SAVEd to tape as and when required. Once the source code is complete, it may then be compiled into the White Lightning dictionary for later execution.

Included in this documentation is a glossary of Fig-Forth terms (courtesy of the FORTH INTEREST GROUP, P.O. BOX 1105, SAN CARLOS, CA 94070).

Spectra Forth was written by Stuart Smith, the author of the extremely successful DRAGONFORTH, and is an enhancement of a program written by the Forth Interest Group - to whom we offer our thanks.

At the time of writing, floppy discs are not readily available for the Spectrum and instructions referring to discs should be interpreted as accessing RAM.

AN INTRODUCTION TO SPECTRA FORTH

This introduction does not set out to teach Forth programming, but rather to serve as a supplement to available texts on the subject; references include:

'Starting Forth' by Brodie, published by Prentice Hall.
'Introduction to Forth' by Knecht, published by Prentice Hall
'Discover Forth' by Hogan, published by McGraw Hill.

White Lightning syntax consists of Forth words or literals, separated by spaces and terminated by a carriage return. A valid name must not contain any embedded spaces since this will be interpreted as two distinct words, and most be less than 31 characters in length. If a word is entered which does not exist or has been spelt wrongly, or the number entered is not valid in the current base, then an error message will be displayed. To compile and execute programs created using the Editor type LOAD <CR>. Throughout these examples <CR> means 'PRESS ENTER'.

```
e.g.  -FINE     will generate an error message 0 since the word does
      not exist.

      HEX 17FZ  will generate an error message 0 since Z is not valid
      in hexadecimal base.
```

Other error messages include:

```
      STACK EMPTY
      STACK FULL
      DICTIONARY FULL
```

In order to program in white Lightning, it is necessary to define new words based on the words already in the vocabulary. Values to be passed to these words are pushed onto the stack and if required, the word will pull these values from the stack, operate on them, and push the result onto the stack for use by another Lightning word. As mentioned previously, Spectra Forth (as with all Forths) uses Reverse Polish Notation and integer numbers, therefore no precedence of operators is available, thus all operations are performed in the sequence in which they are found on the stack.

```
e.g.    1 2 + 3 * is equivalent to 3*(1+2)
```

As can be seen, in RPN, the operators are input after the numbers on which they have to operate have been input.

We will now discuss some of the words in greater depth.

1. INPUT/OUTPUT Operators.

EMIT : This will take the number held on the top of the stack and display it on the terminal, as its original ASCII character.

```
e.g.    HEX  41  EMIT  CR  <CR>
```

will instruct the Forth to move into hexadecimal mode, push 41H onto the stack, and then take that number and display it on the terminal - in this example the character displayed will be an "A". The actual character displayed my be any of the recognisable ASCII characters, a graphic character, or a control code depending on the value of the number on the stack.

EMITC : As EMIT but Control characters are also dealt with.

KEY : This will poll the keyboard, wait for a key to be pressed and push the ASCII code for that key onto the stack, without displaying it on the terminal.

e.g. KEY   Press "A" on the keyboard

will instruct the computer to wait for a key to be pressed (press the "A") and then push the ASCII value of this key, in this case 41H (where the 'H' implies Hexadecimal 41 ie 65 decimal) onto the top of the stack. In order to display this character, try the following example:

Type:

```
KEY EMIT <CR>
```

but be sure to hit the <CR> very quickly.

Now hit any key and its ASCII value will be printed followed by OK. So if you type "A" it would print "AOK". If you were too slow you've now got two cursors! Ignore the top one and try again. This problem only occurs because when you press <CR> to enter the example, it immediately executes and you've possibly still got <CR> held down. In a normal Forth definition you won't have this problem.

CR : This will transmit a carriage return and line feed to the display.

. : Convert the number held on the stack using the current BASE and print it on the screen with a trailing space.

e.g   Suppose the stack contains 16H and BASE is decimal (10), then . will print 22 (this is 16 + 6); if BASE were hexadecimal (16), then . would print 16.

In order to see this Working we will alter the BASE and push numbers onto the stack - remember, that just by typing in a valid number will result in it being pushed onto the stack. There are two words to alter the BASE:

HEX : Use hexadecimal base
DECIMAL : Use decimal base

Try:

(i)   HEX 1F7 . <CR> (Where <CR> means press ENTER).
      This will print 1F7

(ii) DECIMAL 2048 . <CR>
      This will print 2048

(iii) DECIMAL 2048 HEX . <CR>
      This will print 800, since this is the HEX equivalent, of 2048.
      Remember that . will remove the number from the stack that it is printing.

U. : Prints the number held on the top of the stack as an unsigned number.

e.g.   HEX C000 U. <CR>

will push C000 onto the stack and then print it.

If we use just . we will get a negative result.

```
      HEX C000 . <CR>
```

will print -4000

? : Print the value contained at the address on top of the stack using the current base.

Suppose the top of the stack contains FF40H, location FF40/41H contains 0014H, and current BASE is 10 (DECIMAL), then ? will print 20 which is the decimal equivalent of 14 Hex.

TYPE : This uses the top TWO numbers held on the stack and will print a selected number of characters starting at a specific address onto the screen. The top number on the stack is the character count and the second number is the address to start at.

e.g. `HEX 6100 20 TYPE <CR>`
(Note that 6100H is pushed onto the stack and 20H is pushed on top of it 20H = TOP; 6100H = second). This will print 20H (32) ASCII characters corresponding to the data starting at address 6100H . (Note that much of the output will be unrecognisable unless the data contains correct ASCII codes, such as for numbers and letters).

DUMP : This takes the top number on the stack and prints out 80H bytes starting at this address.

" : This is used in the form ." character string " and will display the string contained within " " on the screen.

e.g. `." THIS IS A CHARACTER STRING " <CR>`
will put THIS IS A CHARACTER STRING on the screen. Note the spaces between the string and the quotes.

SPACE : This will display a single blank/space on the screen.

SPACES : This will display n spaces on the screen, where n is the number on the top of the stack.

e.g. `DECIMAL 10 SPACES <CR>`
will print 10 spaces on the screen.

2. MATHEMATICAL OPERATORS

+ : This will add the top two numbers on the stack and leave the result as a single number.

e.g. `1 2 + . <CR>`
will print the value of 1 + 2 = 3 on the screen. Note that the two top numbers are removed from the stack, being replaced by a single number - this is true of most Forth commands, in that they remove the values which they require to use from the stack and push the result onto the stack.

For the purposes of the following examples, let us refer to the numbers on the stack as follows:

```
 N1  =  top    number on stack (i.e. first to be removed)
 N2  =  second number on stack (i.e. second to be removed)
 N3  =  third  number on stack (i.e. third to be removed)
```

To demonstrate this, let us push three numbers onto the stack by typing:

```
        HEX 01FA 0019 1F47 <CR>
```

The stack will look like this:

```
        1F47    Top of stack
        0019
        01FA
```

Note that this illustrates the property of the stack, that it is, Last In First Out or LIFO; therefore we have:

```
        N1  =  1F47
        N2  =  0019
        N3  =  01FA
```

So if we type:

```
        CR . CR . CR . CR <CR>
```

```
We get  1F47
          19
         1FA
```

We will now resume our explanation of the mathematical operators.

```
-       :  This will subtract the top number on the stack from the second number
           on the stack and leave the result as the top number.
i.e.    N1 = N2 - N1

e.g.    Decimal 7 11 - . <CR>
will print -4, since the stack would contain

        N1     11    TOS (Top of stack)
        N2      7

before the subtraction, and

        N1     -4    TOS

after the subtraction.
```

```
*         : This will multiply the top two numbers on the stack and leave the
           result on the top of the stack.
i.e.    N1 = N1 x N2

e.g.    DECIMAL 140 20 * . <CR>
would print 2800
```

```
/       :  This will divide the second number on the stack by the first number,
           and leave the result on the top of the stack.
i.e.    N1 = N2 / N1

e.g.    DECIMAL 1000 500 / . <CR>
will print 2


MAX     :  This will leave the greater of the top two numbers on the stack.

e.g.    371 309 MAX . <CR>
will print 371


MIN     :  This will leave the smaller of the two numbers on the stack.

e.g.    371 309 MIN . <CR>
will print 309


ABS     :  This will leave the absolute value of the top number on the stack as
           an unsigned number.
i.e.    N1 = ABS(N1)

e.g.    47 ABS . <CR>
will print 47

        -47 ABS . <CR>
will print 47


MINUS   :  This will negate the top number on the stack.
i.e.    N1 = -N1

e.g.    418 MINUS . <CR>
will print -418

        -418 MINUS . <CR>
will print 418


1+      :  add 1 to the top number on the stack
        N1 = N1 + 1


2+      :  add 2 to the top number an the stack
        N1 = N1 + 2


1-      :  subtract 1 from the top number on the stack
        N1 = N1 - 1


2-      :  subtract 2 from the top number on the stack
        N1 = N1 - 2

e.g.    196 2- . <CR>
will print 194
```

MOD : This will leave the remainder of N2/N1 on the top of the stack with the same sign as N2

```
e.g.    17 3 MOD . <CR>
will print 2  (17/3 = 5 remainder 2)
```

/MOD : This will leave the remainder and the quotient on the stack of N2/N1 such that the quotient becomes the top number on the stack and the remainder becomes the second.

```
e.g.    17 3 /MOD . CR . <CR>
will print 5 (quotient)
           2 (remainder)
```

3. STACK OPERATORS

DUP : This will duplicate the top number on the stack.

```
e.g.    719 DUP . . <CR>
will print  719 719
```

DROP : This will drop the number from the top of the stack.

```
e.g.    111 222 DROP . <CR>
will print  111
```

SWAP : This will swap the top two numbers on the stack.

```
e.g.    111 222 SWAP . . <CR>
will print  111 222
```

OVER : This will copy the second number on the stack, making it a new number at the top of the stack without destroying the other numbers.

```
e.g.    111 222 OVER . CR . CR . <CR>
will print  111
            222
            111
```

since the stack before OVER was:

```
            222      TOS
            111
```

and after OVER is:

```
            111      TOS
     copy   222
            111
```

ROT : This will rotate the top three numbers on the stack, bringing the third number to the top of the stack.

```
e.g.    1 2 3 ROT . CR . CR . <CR>
will print  1
            3
            2

since the stack before ROT was:
            3        TOS
            2
            1

and after ROT is:
            1        TOS
            3
            2
```

4. OTHER OPERATIONS

!   : This will store the second number on the stack at the address held on the top of the stack. (pronounced "store").

```
e.g.    Suppose the stack is as follows:

        HEX C000     TOS
            FFEE

This will store FFEE at address C000/C001
i.e.    EE at C000
        FF at C001

If we key in HEX FF00 C000 ! <CR>
this will store FF00 at C000/C001
i.e     C000 contains low byte 00
        C001 contains high byte FF
```

Remember that each 16 bit number takes up 2 bytes.

@   : This will replace the address held on the top of the stack, with the 16 bit contents of that address. (Pronounced "at")

Suppose the memory contents are as follows:

```
Address:  6100 6101 6102 6103 6104 6105
Contents: 00   C3   8F   70   00   C3

then 6100 @ . <CR>
will print C300
```

If you wish to deal with single bytes, then a variation of the above will be used.

C!   : Will store a single byte held in the second number on the stack at the address held on the top of the stack.

```
e.g.    FF C000 C! <CR>
will store a single byte FF at address C000.
```

C@   : This will fetch the single byte held at the address at the top of the stack - this single byte will be pushed on the stack as a 16 bit number, but with the high byte set to zero.

with reference to the memory contents shown previously,
if we key in C000 C@ . <CR>

this will print FF (and not FF00 as with @)

+!   : This will add the number held in the second number of the stack, to the value held at the address on the top of the stack (Pronounced "Plus-store").

e.g.   4 HEX C000 +! <CR>
will add 4 to the value at C000/C001
As will be shown later, this is of use when using variables in White Lightning.

5. COLON DEFINITIONS

These are the most powerful and most used forms of data structures in White Lightning, and are so called because they begin with a colon ":"

Colon definitions allow the creation of new Forth words based on previously defined words. They can be of any length, although carriage return must be pressed before a particular section exceeds 80 characters.

The general format is:

```
:  new-word  word1  word2.....  wordn  ;
```

All colon definitions end with a semi-colon ";"

If a word used in a colon definition has not been previously defined, then an error will result.

The new-word is executed simply by typing its name and pressing ENTER.

e.g.   Suppose we wish to define a new word to calculate the square of a given number.

We could do this by:

```
: SQUARE DECIMAL CR ." THE SQUARE OF " DUP . ." IS " DUP * . ; <CR>
```

Here we have defined a new word called SQUARE which will be called by

number SQUARE <CR>

e.g.   9 SQUARE <CR>

will result in:

```
THE SQUARE OF 9 IS 81
```

If we follow the operation of the word, we will see the changes in the stack:

```
    TOS                  OPERATION              RESULT
  empty
      9                    9 SQUARE
      9                    CR                 carriage return
      9                    ."                 THE SQUARE OF
  9   9                    DUP
      9                    .                  9
      9                    ."                 IS
  9   9                    DUP
     81                    *
  empty 81                 .                  81
```

and execution of SQUARE ends at the semi-colon.

If we now wished, we could define a new word using our word SQUARE.

We are now going to discuss control structures. It must be remembered, that the control structures can only be incorporated in colon definitions, or an error will result.

6. CONTROL STRUCTURES

LOOPS

There are essentially two forms of loop operation:

 (i) DO... LOOP

(ii) DO... +LOOP

The first loop structure is used as follows:

```
limit start DO ... 'Forth words' ... LOOP
```

The Forth words within the loop are executed until start = limit, incrementing the start (or index) by one each time. Type:

```
: TEST1 5 0 DO ." Forth " CR LOOP ; <CR>
Typing in TEST1 <CR>
will print Forth
           Forth
           Forth
           Forth
           Forth
```

The second loop structure is used as follows:

```
limit start DO ...'Forth words' increment +LOOP
```

The Forth words within the loop are executed from start to limit, with the index being incremented or decremented by the value increment. Try:

```
: TEST2 5 0 CO ." HELLO " 2 +LOOP ; <CR>
Executing TEST2 will print HELLO HELLO HELLO
```

Since the limit and the index are held on the return stack, it would be useful if we could examine the index. Well, there are words to do this:

I : This will copy the loop index from the return stack onto the data stack.

J : This will push the value of the nested LOOP index to the stack.

K : This will push the value of the double nested LOOP index to the stack.

Type:

```
: TEST3 4 0 DO 4 0 DO 4 0 DO K J I . . . CR LOOP LOOP LOOP ; <CR>
```

```
Executing TEST3      1  1  1
will print:          1  1  2
                     1  1  3
                     .  .  .
                     .  .  .
                     .  .  .
```

and so on.

7. CONDITIONAL BRANCHING

Conditional branching must again be used only within a colon definition and uses the form:

IF (true part) ... (Forth WORDS) ... **ENDIF**

IF (true part) ... (Forth WORDS) ... **ELSE** (false part) ... (Forth WORDS) ... **ENDIF**

These conditional statements rely on testing the top number on the stack to decide whether to execute the TRUE part, or the FALSE part of the condition.

If the top item on the stack is true (non-zero) then the true part will be executed. If the top item is false (zero) then the true part will be skipped and execution of the false part will take place. If the ELSE part is missing, then execution skips to just after the ENDIF statement.

There are several mathematical operators which will leave either a true (non-zero) flag, or a false (zero) flag on the stack to be tested for by IF.

These are:

0< : This will leave a true flag on the stack if the number on the top of the stack is less than zero, otherwise it leaves a false flag.

e.g. `-4 0< <CR>`
will leave a true flag (non-zero).

To see this, type:
```
        . <CR>
```
to print the top number on the stack, which is the flag. This will print
```
        1
```
to show a true flag.

```
        914 0< . <CR>
```
will print a 0 (false flag).

```
0=      :  This will leave a true flag on the top of the stack if the number on
           the top of the stack is equal to zero, otherwise it will leave a
           false flag.


<       :  This will leave a true flag if the second number on the stack is less
           than the top number, otherwise it will leave a false flag.

e.g.    40 25 < . <CR>
will print 0 (false flag).

If we look at the stack during this operation we will see:

            Operation              TOS
               40                   40
               25                 40 25
                <                     0
                .                 empty


>       :  This will leave a true flag if the second number on the stack is
           greater than the top number, else a false flag will be left.

e.g.    40 25 > . <CR>
will print 1 (true flag).


=       :  This will leave a true flag if the two top numbers are equal,
           otherwise it will leave a false flag.

Now for some examples using the conditional branching structures, type:

: TEST= = IF ." BOTH ARE EQUAL " ENDIF ." FINISHED " ; <CR>

Now key in two numbers followed by TEST= and a carriage return.

e.g.    11 119 TEST= <CR>
This will print FINISHED

        119 119 TEST= <CR>
will print BOTH ARE EQUAL FINISHED

Now key in:
: TEST1= = IF ." EQUAL " ELSE ." UNEQUAL " ENDIF CR ." FINISHED " ; <CR>

Now key in:
        249 249 TEST1= <CR>
this will print EQUAL
                FINISHED

Try:    249 248 TEST1= <CR>
this will print UNEQUAL
                FINISHED

Notice how the part after ENDIF was executed in both cases.

Two more loop structures will now be discussed:
```

BEGIN .... (Forth WORDS) .... UNTIL

BEGIN .... (Forth WORDS) .... WHILE .... (Forth WORDS) .... REPEAT

Using the BEGIN .... UNTIL the value at the top of the stack is tested upon reaching UNTIL. If the flag is false (0) then the loop starting from BEGIN is repeated. If the value is true (non-zero) then an exit from the loop occurs.

Try typing the following example:

```
: COUNT-DOWN DECIMAL 100 BEGIN 1- DUP DUP . CR 0= UNTIL ." DONE " ; <CR>
```

Now key in: COUNTDOWN <CR>
This will print:

```
  99
  98
   .
   .
   .
   3
   2
   1
   0
DONE
```

The BEGIN ... WHILE ... REPEAT structure uses the WHILE condition to abort a loop in the middle of that loop. WHILE will test the flag left on top of the stack and if that flag is true, will continue with the execution of words up to REPEAT, which then branches always (unconditionally) back to BEGIN. If the flag is false, then WHILE will cause execution to skip the words up to REPEAT and thus exit from the loop.

We will now construct a program to print out the cubes of numbers from 1 upwards, until the cube is greater than 3000.

The colon definition could be as follows:

```
: CUBE DECIMAL 0 BEGIN 1+ <CR>
DUP DUP DUP DUP * * DUP <CR>
3000 < WHILE ." THE CUBE OF " <CR>
SWAP . ." IS " . CR REPEAT <CR>
DROP DROP DROP ." ALL DONE " CR ; <CR>
```

You may get an error message "MSG#4" appearing on the screen; this means that the word you have just created already exists. This is not a problem since the new word will be created, and all actions referencing the word CUBE will be directed to the latest definition using that name.

Now run this by keying in:

```
CUBE <CR>
```

and watch the results.

Try to follow what is happening by writing down the values on the stack at each operation. If you are having any difficulty in doing this, the stack values are shown below.

STACK	OPERATION	OUTPUT (if any)
empty	DECIMAL	
0	0	
0	BEGIN	
1	1+	

(let us now refer to the number on the stack as N)

N N	DUP	
N N N	DUP	
N N N N	DUP	
N N N N N	DUP	
N N N N^2	*	
N N N^3	*	
N N N^3 N^3	DUP	
N N N^3 N^3 3000	3000	
N N N^3 flag (1 or 0)	<	

If TRUE:

N N N^3	WHILE	
	." THE CUBE OF "	THE CUBE OF
N N^3 N	SWAP	
N N^3	.	N
	." IS "	IS
N	.	N^3
N	CR	carriage return
N	REPEAT	(branch back to BEGIN)

If FALSE:

N N	DROP	
N	DROP	
empty	DROP	
	." ALL DONE "	ALL DONE
	CR	
	;	

In fact, it is a good idea to check the stack contents during the execution of any new Forth word to make sure that it is working correctly. (Note that DROP merely clears the top number from the stack).

Finally, one extra construct has been added to circumvent the problem of deeply nested IF...THEN...ELSE structures. This is the CASE OF structure. It takes the general form :

```
CASE n1 OF (Forth Word) ENDOF n2 OF (Forth Word) ENDOF ... END CASE
```

For example type:

```
: TEST4 CASE 1 OF ." FIRST CASE " ENDOF 2 OF ." SECOND CASE " ENDOF 3 OF ." THIRD
CASE " ENDOF END CASE ; <CR>
```

Now type :

```
1 TEST4 CR 2 TEST4 CR 3 TEST4 CR <CR>
```

8. CONSTANTS AND VARIABLES

White Lightning also allows you to define your own constants and variables using the Fourth words:

CONSTANT
VARIABLE

When a constant is called up, this causes its VALUE to be pushed onto the stack, however, when a variable is called up, this causes its address to be pushed onto the stack. The Forth words ! and @ are used to modify the contents of the variable.

A constant is defined by using the form:

```
value CONSTANT name
```

and any references to the name will cause the value n to be put on the stack.

A variable is defined using the form:

```
value VARIABLE name
```

and any reference to the name will result in the address of that variable to be put on the stack for further manipulation using ! and @. It is essential that you realise the difference between the contents and the address of a variable.

Now for some examples:

```
  64 CONSTANT R 1000 CONSTANT Q
 256 VARIABLE X
   0 VARIABLE Y
```

```
R Q + .  will print the value of R + Q i.e. 1064
    X .  will print the address of X, not its value
  X @ .  will print the value of X, i.e. 256
  R Y !  will store the value of R in the variable Y
  Y X !  will store the address of Y in the variable X
  4 X !  will store the value 4 in variable X
```

BASIC statement	**Forth Equivalent**
LET X = Y	Y @ X !
LET X = R	R X !
LET X = 4	4 X !
LET X = X + 5	5 X +!

OTHER COMMONLY USED FORTH WORDS

LIST : This will list the contents of the screen number held on the top of the stack.

e.g. 6 LIST will list screen 6 to the screen. Note that if source has not been typed into any of the screens, they will probably contain garbage.

FORGET : This is used to delete part of the Lightning dictionary. Please note that not only will the word following FORGET be erased, but so will every word defined after it!

e.g. FORGET EXAMPLE will delete the word EXAMPLE (if it exists) along with any other words defined after it.

VLIST : This is just typed in as a single word with no parameters. It will cause a list of all the words defined so far; pressing BREAK (CAPS SHIFT & SPACE as in BASIC) will stop the listing.

LOAD : This will compile the source code that you have created using the editor into the White Lightning dictionary, to become new Lightning words. Loading will terminate at the end of a screen or at the Forth word ;S unless the "continue loading" word --> is used at the end of a screen. The idea of the screen will became obvious in the next section on editing.

USING THE EDITOR

Generally speaking, most users will want to use the Spectrum editor to type and edit the source, but a full Forth line editor is included for compatibility with existing texts. The maximum length of any line is 64 characters. Any characters after this will be ignored.

Line Editor

Included in this version of White Lightning is a line editor to enable you to create source or text files. To facilitate text editing, the text is organised into blocks of 512 bytes, divided into 8 lines of 64 characters. Once the text has been edited, it may then be compiled into the White Lightning dictionary and the text, if required, can be saved to tape. The text is stored in memory in the pages at C000 to F000, therefore, you can edit into screens 1 to 23. If the background facility is utilised, text is stored from CC00 onwards in screens 6 to 23, and screens 0 to 5 cannot be used.

Here is a list of the editor commands and their descriptions:

H : This will Hold the text pointed to by the top number on the stack of the current screen in a temporary area known as PAD.

e.g. 4 H will hold line 4 of the current screen in PAD.

S : Fill (Spread) the line number at the top of the stack with blanks, and shift down all subsequent lines by 1, with the last line being lost.

e.g. 6 S will fill line 6 with blanks and move all other lines down by one, pushing the last line off the screen.

D : Delete the line number held on the stack. All other lines are moved up by 1. The line is held in PAD in case it is still needed. Line 7 cannot be deleted.

E : Erase the line number at the top of the stack by filling it with spaces.

RE : REplace the line number at the top of the stack with the line currently held in PAD.

P	:	Put the following text on the line number held on the stack, by overwriting its present contents.
INS	:	INSert the text from PAD to the line number held on the stack. The original and subsequent lines are moved down by 1 with the last line being lost.
EDIT	:	Works just like the normal Sinclair line editor. Also, it does an automatic list and an automatic flush. This is far and away the best way to edit and the above are included only for compatibility with existing Forths.
CLEAR	:	Clear the screen number held on the stack and make it the current screen.
WHERE	:	If an error occurs during the loading of White Lightning's text screens, then keying in WHERE will result in the screen number and the offending line being displayed. You can now use the other editing commands to edit the screen, or you may move to another screen by either LISTing or CLEARing it.

e.g. 15 LIST will now make screen 15 the current screen and will list the contents.

In order to compile this screen into the dictionary, it is necessary to use the word LOAD.

LOAD

This will start loading at the screen number held on the top of the stack and will stop at the end of the screen.

If you wish to continue and LOAD the next screen, the current screen must end with -->

This means "continue loading and interpreting".

If you wish to stop the LOADing anywhere in a screen then use: ;S

This means "stop loading and interpreting".

At the end of every editing session, and before saving your text, it is necessary to FLUSH the memory buffers into the text area. To do this, just key in:

FLUSH <CR>

Note that the EDIT command does an automatic FLUSH.

You can save your text to tape using the Spectrum 'SAVE' command. You must first enter BASIC by typing PROG <CR> .

Now for an example of how to edit a text file:

The first step is to either LIST or CLEAR the screen about to be worked on:

```
9 CLEAR <CR>
```

This sets the current screen to 9. To insert text use the EDIT command. Type 0 EDIT <CR> followed by the text below:

```
THIS IS HOW TO PUT <CR>
```

Then type 1 EDIT <CR>

```
TEXT ON LINE 1 <CR>
```

and so on, until you have entered:

```
0 THIS IS HOW TO PUT <CR>
1 TEXT ON LINE 1 <CR>
2 LINE 2 <CR>
3 AND LINE 3 OF THIS SCREEN <CR>
```

9 LIST will produce:

```
SCR # 9
0  THIS IS HOW TO PUT
1  TEXT ON LINE 1
2  LINE 2
3  AND LINE 3 OF THIS SCREEN 4
5
6
7
```

To change LINE 2, type 2 EDIT <CR> and then change it in the normal way to insert 'TEXT ON' before 'LINE 2'. Now type 9 LIST <CR> to see the result. The editor ignores characters after the 64th character of the line being edited.

If you have a Sinclair printer connected, then it is probably worth defining a word to list screens to the printer:

```
: SLIST PRT-ON 1+ SWAP DO I LIST CR LOOP PRT-OFF ;
```

To use the above word, type the first screen number, last screen number, SLIST.

e.g. 6 9 SLIST <CR> will list screens 6-9 to the printer.

FORTH ERROR MESSAGES

The following error messages may occur, and will be printed out in the form FRED ? MSG #0   standing for FRED ? ERROR MESSAGE NUMBER 0 .

# 0 - this means that a word could not be found, or that a numeric conversion could not take place.

e.g. 109Z <CR>

# 1 - this indicates an empty stack and will be encountered when trying to take more values from the stack than exist. Try:

```
        : TEST1  1000  0  DO  ?STACK  DROP LOOP  ; <CR>
          TEST1 <CR>
```

?STACK is a word which tests the stack for out of bounds.

# 2 - this indicates that either the dictionary has grown up to meet the stack (dictionary full) or that the stack has grown down to meet the dictionary.

```
Try:    : TEST2  1000  0  DO  ?STACK  0 0 0 0 0 LOOP  ; <CR>
         TEST2 <CR>
```

# 4 - this means that you have redefined an existing word using a new colon definition

```
Try:    : ROT ." NEW DEFINITION " ; <CR>
```

This is not really an error since the new word is still valid, but the old definition cannot be accessed unless you FORGET the new one.

# 6 - this error my occur when editing, loading or listing screens of data.

```
Try:    25 LIST <CR>
```

This will produce MSG#6 and means you have tried to access a non-existent screenful of memory.

# 9 - this indicates that an attempt was made to clear sprite space of less than 2 bytes.

# 10 - this indicates that one of the IDEAL words made reference to a sprite which did not exist, or that an attempt was mode to insert a sprite using ISPRITE with a number previously allocated to an existing sprite.

# 17 - this will occur if you try to use a word in the 'immediate' mode which should only be used during compilation, i.e. during colon definitions. For a list of such words, refer to the glossary (words with "C" in the top right hand corner of the description).

```
Try:    DO <CR>
        IF <CR>
```

# 18 - this occurs if a word meant for execution only, is put within a colon definition (words with "E" in the top right hand corner of the description).

# 19 - this means that a colon definition contains conditionals that have not been paired.

```
e.g.    a LOOP without a DO
        an ENDIF without an IF
```

```
Try:    : TEST3  ELSE  ." WRONG " ; <CR>
```

# 20 - this occurs if a colon definition has not been properly finished.

```
Try:    : TEST4 IF ." OK " ; <CR>
```

# 21 - this means that you have tried to delete something in the protected part of the Forth dictionary, e.g.

```
FORGET DO
```

# 22 - this implies the illegal use of --> when not loading text screens.

# 23 - this happens when you try to edit a non-existent line of screen data.

Try:

```
12 D
```

SECTION 3

IDEAL
by John Gross

IDEAL has been designed to facilitate the manipulation of sprites and screen data, and with its 100 or so instructions, provides a powerful and comprehensive animation sub-language. Time should be taken to gain familiarity with the available commands before undertaking the first big project. Remember, that by using the colon definitions of Forth, new words can very easily be added to the language, built from the existing Forth and graphics words. Mastering this technique effectively will save a great deal of space and, in many cases, execution time.

SPRITE

A sprite is a software controllable graphics character. White Lightning supports up to 255 sprites with user selectable dimensions.

SCREEN WINDOWS

A screen window is a section of the screen defined by the four variables COL, ROW, HGT and LEN. Columns are in the range 0 to 31, Rows are in the range 0 to 23, Heights are in the range 1 to 24, and Lengths are in the range 1 to 32. The unit for each is the character. COL and ROW specify the position of the top left hand corner of the window, with ROW 0 at the top of the screen and COL 0 on the left hand side of the screen. HGT and LEN define the size of the window. To see an example type:

```
5 ROW ! 6 COL ! 4 HGT ! 3 LEN ! INVV <CR>
```

The window has been inverted to mark it out.

SPRITE WINDOWS

A sprite window is a section of the sprite defined by the Forth variables SCOL, SROW, HGT and LEN. This time SCOL and SROW specify the position of the top left hand corner of the sprite window. HGT and LEN are again used to specify the dimensions of the window.

SPRITE SPACE

Sprite space is the area of memory containing the previously defined sprites. The variable SPST holds the address of the start of sprite space, so SPST should never be loaded with a new value unless a COLD# command is being executed, or you're quite sure you know what you're doing!! SPND points to the first free byte after sprite space. SPND should never be higher than FFF0 Hex if routines are being executed in Background.

PIXEL DATA

For those not acquainted with the workings of the Spectrum screen display, each character on the screen is produced as follows: each character cell is an array of 64 (8 by 8) pixels. A pixel is a 'dot' which can be INK colour or PAPER colour. The bytes which define a particular character or block of characters are referred to as pixel data.

ATTRIBUTE DATA

The colour of the INK and PAPER in each particular cell, together with the BRIGHTNESS and FLASHING attributes of the character, are controlled by a separate byte. The bytes which define the attributes of the block of characters are referred to as Attribute data. Pixel data and Attribute data are frequently treated as separate entities.

SCREEN OPERATIONS

Often, it is required to carry out an operation, such as a scroll or a reflection, on one particular section of the screen. Four variables are used to define a screen window, these are COL, ROW, HGT and LEN. The co-ordinates of the top left hand corner are held in COL and ROW, where COL is measured from the left and ROW from the top. Both values are in characters. HGT and LEN are the dimensions of the window. COL + LEN must be in the range 1 to 32, and ROW + HGT must be in the range 1 to 24. Commands in this group are postfixed with a "V", e.g. WRL1V, INVV, MIRV.

SCREEN/SPRITE 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 of the window. If the window overlaps the edge of the screen, the command will not execute. Typical commands in this group are the PUTs and GETs, which move sprites between the screen and memory. Commands in this group are postfixed with an "S", e.g. PUTBLS, GETXRS.

SPRITE OPERATIONS

These cover more or less the same commands as the screen operations, but this time a complete sprite is used instead of a screen window. The only parameter required is the sprite number stored in SPN. Commands in this group are postfixed with an "M", e.g. WRR4M, ATTUPM.

SCREEN/SPRITE WINDOW 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, SCOL and SROW are used to define the position of the window within the sprite. SCOL and SROW are measured in characters, SROW from the top and SCOL from the left. If SROW + HGT is greater than 24 or the sprite height, or if SCOL + LEN is greater than 32 or the sprite width, the commands will not execute. These commands are postfixed with an "S", e.g. GWATTS, PWORS.

SPRITE/SPRITE WINDOW OPERATIONS

These are operations between a whole sprite and a sprite window. The two sprite numbers are held in SP1 (the whole sprite) and SP2 (the sprite which contains the window). The dimensions of the window are the dimensions of the sprite whose number is held in SP1. The position of the window in the sprite whose number is held in SP2 is specified by SCOL and SROW. Commands in this group are postfixed with an "M", e.g. GWATTM, PWNDM.

SPRITE/SPRITE OPERATIONS

These are operations between sprites which usually have the same dimensions, or, as in the case of the SPIN command, transposed dimensions. SP1 and SP2 hold the sprite numbers. Commands in this group are postfixed with a "M", e.g. COPORM, SPINM.

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, or maybe a collision detection sprite.

IDEAL VARIABLES

The IDEAL sublanguage uses 27 variables in all, these are:

VARIABLE	USE
ROW	Used to hold the row (Y co-ord) in characters, measured from the top of the screen (0-23).
LEN	Used to hold the width of the current screen window (1-32), or the width of the sprite being defined (1-255). Units are characters.
COL	Used to hold the column (X co-ord) in characters, measured from the left of the screen (0-31).
HGT	Used to hold the height of the current screen window (1-24), or the height of the sprite being defined (1-255). Units are characters.
SROW	Used to hold the row (Y co-ord) within the sprite whose number is held in SP2, measured from the top (0-(HGT-1)). Units are characters.
SCOL	Used to hold the column (X co-ord) within the sprite whose number is held in SP2, measured from the left (0-(LEN-1)). Units are characters.
NPX	Used to hold the size and direction of the vertical scrolls. Positive scrolls are upward and negative downward. Units are pixels and not characters.
SPN	Used to hold the sprite number for those words which operate on only one sprite (1-255).
SP1	Where operations involve a sprite and a sprite window, SP1 holds the number of the sprite which does not contain the window (1-255). Where a sprite is to be spun into a second sprite, SP1 holds the number of the first sprite (1-255).
SP2	Where operations involve a sprite and a sprite window, SP2 holds the number of the sprite which does contain the window (1-255). Where a sprite is to be spun into a second sprite, SP2 holds the number of the second sprite (1-255).

SPST	Used to hold the start address of sprite space.
SPND	Used to hold the end of sprite space, i.e. the first free byte after the last sprite. This is the address of the foreground scrolling buffer.
SLEN	Used to hold the length of sprite space to be cleared by the COLD# command.
MLEN	Used to hold the size and direction of the relocation. A positive value relocates sprites to higher memory and a negative value to lower memory.
SPTR	On return from the TEST command, SPTR points to the start of the sprite.
DPTR	On return from the TEST command, DPTR points to the start of the pixel data.

Alternate Variables

Eleven of the previously listed variables are replicated for use by the background program (see Foreground/Background). These are ROW', COL', LEN', HGT', NPX', SPN', SP1', SP2', SROW', SCOL' and SPND'.

When a word is executed in background, the eleven alternate variables are automatically switched with the eleven background variables; when execution is complete, the variables are switched again to restore them to their former state.

Suppose, for example, that the background program is to scroll left 1 pixel with wrap (WRL1V), with an area of screen 6 characters wide and 4 characters high, with top left co-ordinates row = 5, column = 7.

Now type the following:

```
CLS 6 LEN' ! 4 HGT' ! 5 ROW' ! 7 COL' ! ' WRL1V INT-ON <CR>
```

The window is now scrolling but you can't see it, because there is no data in the window.

Type VLIST <CR> and watch the data as it scrolls through the window. The data in the window will be slanting to the left, because the foreground program was scrolling up at the same time as the background program scrolled left.

Leaving the background program running, type:

```
10 LEN' ! <CR>
```

and the window will widen.

Type:

```
INT-OFF ' WRR8V INT-ON <CR>
```

and the screen will scroll to the right, this time much more rapidly. Now type:

```
INT-OFF <CR>
```

to halt the background program.

In the above example we set background variables from foreground. If we were to set the background variables actually in the background program, then foreground and background variables would already have been switched before execution. To set up the same windows, we would now have to use ROW, COL, HGT and LEN, and not ROW', COL', HGT' and LEN'.

To define a word to do this, type:

```
: FRED 6 LEN ! 4 HGT ! 1 ROW ! 2 COL ! WRL1V ; <CR>
```

To run "FRED" in background, type:

```
' FRED INT-ON <CR>
```

Since the variables were this time being assigned values in the background program itself, the alternate variables set was being accessed with the normal names. Now type:

```
INT-OFF FORGET FRED <CR>
```

to halt the background program and clear the definition.

Operating in this way, a word will work in foreground or background without any need to change variable names. The alternate variables are only used directly by a foreground program that is required to change background variables, or a background program that is required to change foreground variables. If the previous example is a little confusing at first, carry out your own experimentation until it becomes clear.

ERRORS

The graphics commands do not in many cases provide the user with error messages, but instead, if an attempt is made to execute a command which is not possible, for instance scrolling a screen window which lies partly off the screen, the command will simply not execute. This does have the advantage that the user is freed from testing edge conditions, but does mean that a little extra care needs to be exercised. See the words ADJM and ADJV. Errors are generated if an attempt is made to access a non-existent sprite, or to insert an already existing sprite using ISPRITE.

SPRITE AND BUFFER ORGANISATION

Before discussing the sprite manipulation commands in detail, it is worth describing the organisation of sprites in some detail. The user does not need this information, but it is made available for interest and an overall appreciation of the language structure.

Sprites are stored as one contiguous block of data whose start address is held in the variable SPST. The first free byte after the final sprite contains a zero and this address is held in the variable SPND. The format of each sprite is as follows:

First byte	Holds the sprite number which must be in the range 1 to 255.
Second and third bytes	Hold the address of the start of the next sprite.

Fourth byte	Holds the width of the sprite in characters (1 to 255).
Fifth byte	Holds the height of the sprite in characters (1 to 255).
8*height*length bytes	Pixel data.
Height*length bytes	Attribute data.

This means that the total space allocated to each sprite is 9*height*length+5 bytes.

Sprite numbers do not need to run sequentially, but the earlier a sprite is defined, the more rapid its access.

LOADING SPRITES FROM TAPE

Sprites saved to tape using the development software can be loaded into the main program at the start of the session when the "LOAD SPRITES Y/N" prompt appears. If sprites are loaded in this manner, the sprite data, together with the necessary pointers, will be loaded. SPST and SPND are automatically set and the sprites will be ready for use.

If sprites are saved and later loaded from White Lightning, SPST and SPND will need to be set by hand.

THE BUFFER

When vertical scrolling takes place, be it for pixel data or attributes, with or without wrap, data has to be temporarily stored for later retrieval. If a vertical scroll is executed by the foreground program then the buffer is pointed to by SPND, so the space immediately above sprites is used. When the sprite development software is used, a prompt is issued at the start of the session, which asks the user whether or not buffer size should be changed. If the buffer size is not changed then it remains 256 bytes long. The user can enter a larger or smaller value if preferred, though the default value of 256 will cover most eventualities.

Scrolling attributes uses one byte for each column of the width, scrolling pixel data uses one byte for each column of the width, multiplied by the number of pixels being scrolled (the value held in NPX, see vertical scrolls). The buffer space need only be large enough to accommodate the largest scroll, as foreground scrolls will not take place simultaneously. Suppose a sprite or screen window 8 characters high by 4 characters wide is to be scrolled by 10 pixels. (The direction, i.e. the sign of NPX does not matter). The space required is 4 * 10 = 40 bytes. If you find at some later stage that you have not allowed enough buffer space, you can always relocate sprite space downward and likewise, if you have more than you need, you can relocate upwards.

BACKGROUND SCROLLING

When programs are executed in background (see Foreground/Background) it is risky to share a common scrolling buffer, since the background program could execute while the foreground program is using the buffer. For this reason, a second buffer pointer is used for background scrolling. The variable holding the address of the background buffer is SPND'. When White Lightning is first entered, SPND' points to the 256 free bytes in the printer buffer at decimal 23296. The user can move this buffer by changing the value held in SPND'. It is not a bad idea to allocate enough buffer space, for both foreground and background scrolling above sprite space and assign SPND' to point to the space after the foreground buffer. Suppose, for example, the foreground program requires 200 bytes and the background 300 bytes, with the buffer currently set to 256 bytes. 500 bytes are needed in all, so sprites need to be relocated down by 500 - 256 = 244 bytes. Type:

```
-244 MLEN ! RELOCATE <CR>
```

Note that MLEN is now negative since relocation is downward. SPND' should be set 200 bytes into the buffer to leave space for the foreground data. To do this type:

```
SPND @ 200 + SPND' ! <CR>
```

If memory is really tight and the buffer has to be shared, then the background program can be temporarily disabled using DI but as soon as the vertical scroll is executed, an EI must be executed to re-enable the background program. If, for example, a screen window 12 characters wide and 4 characters high is to be scrolled vertically by 8 pixels with wrap, and the background program is to be inhibited, type:

```
0 0 AT 8 NPX ! 12 LEN ! 4 HGT ! 4 COL ! 4 ROW ! DI WCRV EI <CR>
```

It is best to re-enable the background program as soon as possible, preferably, as above, the next word.

Until you get used to the package leave the buffers as they are on entry to White Lightning. Use ISPRITE and DSPRITE, and not SPRITE and WIPE, to define new sprites. The only time you really need to worry about changing buffer sizes or positions is when you have a dire need to save a few extra bytes.

IDEAL MNEMONICS

To get the best out of the White Lightning package, please read these next sections carefully and note the parameters. The words have been selected so as to be as mnemonic as possible. To help yourself become acquainted with the language, it is worth noting the following:

1. Words which involve only the screen are postfixed "V" for "Video Operations".

2. Words which involve only operations on or between sprites, are postfixed with "M" for "Memory Operations".

3. Words which involve operations between the screen and sprites are postfixed with "S" for "Screen/Sprite" operations.

4. BLS indicates that data is being "Block Shifted" to a destination and will replace whatever was there.

5. ORS indicates that data is being "Shifted and OR'ed" so the destination data will be OR'ed with the source data.

6. NDS indicates that data is to be "Shifted and AND'ed".

7. XRS indicates that data is to be "Shifted and XOR'ed".

8. ATT indicates that the operation is on attribute data.

9. WR implies that the data will be scrolled with wrap.

10. SC implies that data will be scrolled without wrap.

11. GW "Get Window" implies that data is being moved from a window into a sprite.

12. PW "Put Window" implies that data is being put into a window from a sprite.

13. COP implies an operation between two sprites with the same dimensions.

The best way to become familiar with the language is to use it!

There are also some general points worth noting.

GENERAL POINTS

1. Vertical scrolls will require some buffer space at the end of sprite space, so make sure that either you have set up sprite space with the development package (the default of 256 is usually adequate), or that at least one COLD# has been executed to make space. The space required for a scroll is obtained by multiplying the width of the sprite, by the number of pixels to be scrolled. Horizontal scrolls do not require buffer space. When White Lightning is first Loaded without sprites, a COLD# is automatically executed and sets the buffer to 256 bytes.

2. All attribute scrolls are "with wrap".

3. Commands prefixed with GET or PUT are operations between a whole sprite and a screen window. These are very fast and can be made even faster by suppressing the movement of attribute data, if its transference is not required. To suppress attribute data use the word ATTOFF. If you wish to switch data flow back on at a later stage, use ATTON. The switch remains in its state until changed by the execution of one of these two words or the execution of a word beginning with GW or PW and ending with S (group 2 GETS and PUTS). NEVER assume the state of the switch at the start of the program: one of your first words at the beginning of your program should be ATTON or ATTOFF.

4. If a sprite is dynamically allocated space at runtime it will probably contain garbage in its pixel and attribute data, so both will need to be set up. It is all too easy to forget the attributes.

5. Sometimes the dynamically created sprites will contain zeroes; if you set up the pixel data and forget the attribute data then execute a PUT to the screen with the attributes on - a black rectangle will appear. See SETAM.

6. If you want to wipe a sprite off the screen but not affect any other pixel data within its screen window, use PUTXRS. Remember, though, that if you have carried out any operations on the original sprite since doing the original PUTBLS or PUTXRS it may not work.

7. If you want to leave a sprite on the screen but clear all other pixel data in its screen window use PUTNDS. Again, be sure that no intermediary operations have taken place.

8. If you wish to PUT a sprite onto the screen over the top of the existing data in the window, then you should use PUTORS.

9. Points 6-8 apply to operations in memory, although the words used are of course different.

10. The best way to get to knew the PUTs and GETs is to experiment with them; you will soon realise how to move sprites. For those of you not yet familiar with what "AND", "OR" and "XOR" mean, note the following:

    a) If two sprites are AND'ed, then only those pixels set in both will remain set when the sprites are AND'ed together.

    b) If two sprites are OR'ed, then all those pixels set in either sprite will be set in the result.

    c) If two sprites are XOR'ed, then all those pixels set in either sprite will be set in the result, but this time, all pixels where both were set will now be reset.

    d) All "Block Shift" commands will destroy whatever was previously in the window.

11. It is possible to set up "masks" in dummy sprites and use the boolean operations OR, XOR and AND to move windows around etc.

12. If you wish to make something appear at lightning speed, leave the data on the screen and fill the window with zero attributes or attributes with the same INK and PAPER colour. To make the sprite appear you need only download the attributes using PWATTS; to make it disappear use PWATTS again, but this time, download stored attributes with the same ink and paper colour, or use SETAV.

13. It is possible to use the TEST command to gain direct access to the attributes in the sprite memory. They are located at DPTR+ 8*LEN*HGT and can be easily block filled.

14. Most animation routines use only the variables:

HGT	Height of window
LEN	Width of window
COL	Screen column of top left character
ROW	Screen row of top left character
SCOL	Sprite column of top left character
SROW	Sprite row of top left character
SPN	Sprite number
SP1	Number of first sprite in a double sprite operation
SP2	Number of second sprite in a double sprite operation

    Columns are measured in characters (8 pixels by 8 pixels) and are counted from the left, 0 to 31.

    Rows are measured in characters and are counted from the top 0 to 23.

    Remember, never change a variable unless you need to, they are not reset between instructions (with the exception of ADJM and ADJV, and you can always write your code to order the operations in such a way as to minimise the resetting of variables.

15. If a word ending in V (screen operation) does not execute, it is almost always because the window you have defined does not lie wholly on the screen. That is to say, that COL + LEN is not in the range 1 to 32, or ROW + HGT is not in the range 1 to 24. See ADJV.

16. If a word ending in S (screen/sprite operation) does not execute, then again, it is almost always because the sprite width + COL is not in the range 1 to 32 or the sprite height + ROW is not in the range 1 to 24. See ADJM.

17. If a word ending in S that is also a window command (second letter W) does not execute, it may be for the reasons outlined in 16, or it may be that SCOL + LEN or SROW + HGT do not lie within the width and height of the sprite containing the window.

18. If a word ending in M that is not a window command does not execute, the sprite probably does not exist; if it is a window command, then it is likely that the width of the first sprite + ROW or the height of the first sprite + SCOL do not lie within the width and height of the sprite containing the window.

19. Remember that the SPIN command needs a second sprite to rotate into, and that its dimensions should be the reverse of the sprite to be spun. For real speed, it is best to store the sprite in each of its 4 orientations.

20. If you wish to do a vertical mirror, just SPIN, do a horizontal mirror, and SPIN back.

21. It is good policy always to make the sprite one character higher and wider than the graphic character itself; this will enable you to scroll the character within the sprite and give pixel resolution when using the PUT commands.

22. If memory permits, it is a good idea to keep a "back-up" of each sprite held under a different number, so that if an error is made and a sprite corrupted, it can be copied back from the copy. They can always be deleted from the final program.

23. Never execute the NEW or CLEAR commands when in BASIC.

24. If a sprite driven under interrupt disappears from the screen or flickers, it is probably a timing related problem and re-ordering the code will almost certainly solve the problem.

25. Screens can be moved up and down memory using the CMOVE word. Each screen occupies 512 bytes and the addresses are listed in Table 1, the Table of Screen Addresses.

26. If you are using the Background facility, the top 16 bytes of RAM will be used, so any data stored there will be corrupted. To avoid this, make sure you have enough scrolling buffer and keep the top of sprite space below 65520.

27. If you execute SETAM or SETAV in background, set the FLASH attribute in the background program.

28. If you wish sprite space to "grow" upward use SPRITE and WIPE, otherwise always use ISPRITE and DSPRITE for sprite allocation.

29. BAD RAMTOP error, usually means you have not RESERVED sufficient space for your BASIC program.

SPRITE UTILITIES

All the sprite utilities described in this section are available at run-time, but we strongly recommend that all sprite allocation is undertaken at the sprite development stage to save laboriously reloading sprite data if an error is made at run-time. They are provided for advanced programming applications only and should never be executed in background.

COLD

This command sets the end of sprite space pointer SPND, to the value in SPST, the start of sprite space pointer. It then clears memory above SPST. The size of the memory cleared is specified in SLEN. Each time a sprite is allocated space using the SPRITE command, SPND is updated. Vertical scroll data (pixel and attribute) uses the space immediately above SPND so a COLD# is necessary at some stage before executing the commands. The amount of data required is given by the product of the scroll width and the number of pixels scrolled. sprites are stored in the following format:

```
First byte is the sprite number              (1 to 255)
Second and third bytes hold the address of the next sprite in memory.
Fourth byte is the sprite width              (1 to 255)
Fifth byte is the sprite height              (1 to 255)
The next 8*height*length bytes hold pixel data.
The final height*length bytes hold the attributes.
Therefore, each sprite requires 9*height*length+5 bytes.
```

It should be noted that sprite numbers need not be allocated in any particular order. The best position for sprite space is at the top of memory and it is not difficult to calculate space required, although the development package does this automatically.

To calculate the total sprite space that will be used, use 9*length*height+5 for each sprite and then add buffer space for vertical scrolls, 256 should be sufficient for most applications. This total should be assigned to SLEN. SPST should then be assigned with 65520-SLEN.

Example:

Suppose you wished to allocate space for 10 4 by 4 sprites, 3 8 by 6 sprites and a 5 by 4 sprite. Suppose also that a 4 pixel scroll will be required for a 4 character wide sprite, and a screen window 9 characters wide will need to be scrolled 3 pixels. The 8 by 6 sprite is also to be rotated. The following procedure to calculate sprite space is required.

```
10  4 by 4 sprites require      10*(4*4*9+5) = 1490 bytes
 3  8 by 6 sprites require       3*(8*6*9+5) = 1311 bytes
 1  5 by 4 sprite requires       1*(i*4*9+5) =  185 bytes
                                      TOTAL  = 2986 bytes
```

The 4 character wide sprite scrolling 4 pixels would require 16 bytes.
The 9 character wide screen window scrolling 3 pixels would require 27 bytes.
If we allow 27, then this will cover both eventualities and we can forget the 16 bytes for the first case.

```
                                      TOTAL  = 27 bytes
```

A dummy sprite 6 by 8 is also required for the rotation so:

```
 1  6 by 8 sprite requires       1*(6*8*9+5) =  437 bytes

So the overall total is          2986+27+437 = 3450 bytes
```

If this is to be located at the top of memory, then SPST will need to be set to 65520 - 3450 = 62070. Note that memory 65520 to 65536 is used for background applications.

In this case you would use the following:

```
62070 SPST ! 3450 SLEN ! COLD#
```

SPND will be automatically set to 62070.

If all this seems a bit complicated, don't worry. A far simpler way of setting up sprites is to use the ISPRITE and DSPRITE commands described later in this section. You won't even need to execute a COLD# command.

SPRITE

Once the sprite space has been cleared, the sprites themselves can be set up.

Parameters	Use
SPN	Number of the sprite to be set up
HGT	Height of the sprite in characters
LEN	Width of the sprite in characters

Command	Action
SPRITE	The five byte leader is set up and SPND adjusted

Note:

If space is dynamically allocated to a sprite, the sprite will not necessarily be initialised and may contain garbage. If a sprite is being set up at run-time, be sure that sufficient memory is available. If a sprite number is given that has been previously used, the old sprite is destroyed and recreated with the new dimensions. If sufficient memory is not available, then either an old sprite can be destroyed or the whole of sprite space can be relocated downwards if space is available. Most users will probably not use the SPRITE command, but instead, will use the far simpler ISPRITE command.

WIPE

This command will destroy the sprite whose number is held in SPN, relocate the sprites above it downward, update the variable SPND (marking the end of sprite space) and leave the particular sprite number free for reallocation.

Parameter	Use
SPND	The number of the sprite to be wiped

Command	Action
WIPE	Destroy a sprite and adjust memory

RELOCATE

An alternative method for creating more space to define new sprites, is to relocate sprite space downward. Only one parameter is required, MLEN. A positive value in MLEN will relocate sprites to high memory and a negative value will relocate them downward. All pointers are reset.

Parameter	Use
MLEN	Size and direction of relocation

Command	Action
RELOCATE	Relocate all sprite data and reset pointers

Example:

If space for a 4 by 4 sprite is to be made and the existing buffer space maintained, the sprites would need to be relocated downward by 4*4*9+5 = 149. To do this use the following:

```
-149 MLEN ! RELOCATE
```

TEST

For advanced applications there is a command to interrogate sprite details. SPN is loaded with the number of the sprite to be interrogated, and after execution, the following parameters will be set:

HGT	will hold the height of the sprite
LEN	will hold the width of the sprite
SPTR	will hold the address of the first byte of the sprite header
DPTR	will hold the address of the first byte of the pixel data
SIZE	will hold the amount of memory occupied by the sprite

In order to calculate the start of attribute data, use DPTR+8*HGT*LEN. Note that HGT, LEN, SPTR and DPTR will all be zero if the sprite wasn't found.

A true or false flag is also placed on the stack. True means the sprite exists and false means it doesn't.

Examples:

To see if sprite 34 exists, and print its dimensions if it does - use:

```
34 SPN ! TEST IF HGT ? LEN ? CR ENDIF
```

ISPRITE AND DSPRITE

If sprites are located at the top of memory, then ISPRITE and DSPRITE can be used to create and destroy sprites without the need for any complex calculations. ISPRITE will insert a sprite at the top of memory, maintain the buffer space and automatically relocate the rest of sprite space downward. DSPRITE will destroy an old sprite and relocate sprite space upward. The easiest way to use this package is to leave the buffer space as it is, then just use ISPRITE and DSPRITE to CREATE and DESTROY sprites. SPRITE, WIPE, COLD# and RELOCATE are for advanced applications where sprites are created during program execution. If a "ZAPPED" program were to execute ISPRITE the sprites would extend downwards and could overwrite the object code of the main program which lies just beneath SPST. If a program runs before being "ZAPPED", but the final run-time version crashes, this is where to look!

Parameter	Use
SPN	Number of the sprite to be inserted
HGT	Height of the sprite to be inserted
LEN	Length of the sprite be be inserted

Command	Action
ISPRITE	Create new sprite and adjust memory

Parameter	Use
SPN	Number of sprite to be inserted

Command	Action
DSPRITE	Wipe old sprite and adjust memory

Note:

Be sure sufficient memory is available before executing ISPRITE. SPST holds the start; after execution, SPST will became SPST - 9*HGT*LEN-5.

SCROLLING

SCREEN SCROLLS

The horizontal screen scrolls are by 1, 4 or 8 pixels, left or right and with or without wrap. The vertical scrolls are slightly more flexible. The variable NPX is loaded with the number of pixels to be scrolled, positive for upward movement and negative for downward movement. In each case, a screen window has to be defined.

Horizontal Scrolls

Parameter	Use
COL	Column of the left hand edge of the window (0-31)
ROW	Row of the top edge of the window (0-23)
LEN	Width of the window (1-32)
HGT	Height of the window (1-24)

Command	Action
WRL1V	Scroll left 1 pixel with wrap
WRR1V	Scroll right 1 pixel with wrap
SCL1V	Scroll left 1 pixel, no wrap
SCR1V	Scroll right 1 pixel, no wrap
WRL4V	Scroll left 4 pixels with wrap
WRR4V	Scroll right 4 pixels with wrap
SCL4V	Scroll left 4 pixels, no wrap
SCR4V	Scroll right 4 pixels, no wrap
WRL8V	Scroll left 8 pixels with wrap
WRR8V	Scroll right 8 pixels with wrap
SCL8V	Scroll left 8 pixels, no wrap
SCR8V	Scroll right 8 pixels, no wrap

Note:

Before executing any of these commands, the window needs to be set up using the four parameters above. If the command does not execute, it is likely that part of the window does not lie on the screen. COL + LEN should be in the range 1 to 32 and LEN + HGT should be in the range 1 to 24.

Example:

To see these commands work, it is a good idea to do a VLIST first so that there is some data on the screen. The four parameters will need to be set, unless of course they have previously been defined, but for this example let us assume that they haven't. To scroll a window at COL 4, ROW 5 with height 4 characters and width 3 characters, with wrap, 4 pixels left, type the following:

```
4 COL ! 5 ROW ! 4 HGT ! 3 LEN ! WRL4V <CR>
```

If there is no data on the screen, type:

```
VLIST 4 COL ! 5 ROW ! 4 HGT ! 3 LEN ! WRL4V <CR>
```

A window column 4 Row 5 has been scrolled left 4 pixels - if you didn't see it happen type:

```
WRL4V WRL4V WRL4V
```

and you should see it!

Parameters do not reset after the execution of the commands, so to repeat the above, this time at column 10 and without wrap, you could use the following:

```
10 COL ! SCL4V <CR>
```

Vertical scrolls

These work in a similar way to the horizontal scrolls, but in addition to setting up the window with the four window parameters COL, ROW, HGT and LEN, a further variable NPX is used to give the size and direction of the scroll in pixels. A positive value for NPX causes upward scrolling and a negative value causes downward scrolling.

Parameter	Use
COL	Column at left hand edge of window (0-31)
ROW	Row of top left edge of the window (0-23)
LEN	Width of the window (1-32)
HGT	Height of the window (1-24)

Command	Action
WCRV	Vertical scroll with wrap
SCRV	Vertical scroll, no wrap

Note:

All vertical scrolling of pixel data and/or attributes for screen or sprites, requires buffer space. The space required is calculated by multiplying NPX by LEN. The start of the buffer is taken as the next free byte after sprite space, so be sure that either a COLD# has been executed, or that SPND has been set. It is safe practice to develop sprites using the software provided, and to allow 256 bytes buffer at the top before linking with the main program. Be sure also that the scroll length is not greater than the window's size. On entry to White Lightning a COLD# is automatically executed and a scrolling buffer of 256 bytes is set up. This is maintained so long as ISPRITE and DSPRITE are used in preference to SPRITE and WIPE.

Attribute Scrolls

Attribute scrolls are similar to the pixel data scrolls but all scrolls are always by one character, with wrap.

Parameter	Use
COL	Column of the left hand edge of the window (0-31)
ROW	Row of the top edge of the window (0-23)
LEN	Width of the window (1-32)
HGT	Height of the window (1-24)

Command	Action
ATTLV	Scroll attributes left 1 character with wrap
ATTRV	Scroll attributes right 1 character with wrap
ATTUPV	Scroll attributes up 1 character with wrap
ATTDNV	Scroll attributes down 1 character with wrap

SPRITE SCROLLS

The format for these commands is similar to that for the screen scrolls, except that these commands are postfixed with an M as opposed to a V.

Horizontal Scrolls

Parameters	Use
SPN	The number of the sprite to be scrolled (1 to 255)

Command	Actions
WRL1M	Scroll left 1 pixel with wrap
WRR1M	Scroll right 1 pixel with wrap
SCL1M	Scroll left 1 pixel, no wrap
SCR1M	Scroll right 1 pixel, no wrap
WRL4M	Scroll left 4 pixels with wrap
WRR4M	Scroll right 4 pixels with wrap
SCL4M	Scroll left 4 pixels, no wrap
SCR4M	Scroll right 4 pixels, no wrap
WRL8M	Scroll left 8 pixels with wrap
WRR8M	Scroll right 8 pixels with wrap
SCL8M	Scroll left 8 pixels, no wrap
SCR8M	Scroll right 8 pixels, no wrap

Note:

The chief purpose of these commands is to give pixel resolution to the PUT command, and for this reason, a 1 character border along 2 edges of a sprite character should always be allowed.

Example:

To scroll sprite number 7, 1 pixel right with wrap, use:

```
7 SPN ! WRR1M
```

Vertical Scrolls

These work in the same way as the vertical screen scrolls where the signed variable NPX is used to determine the size and direction of the scroll.

Parameters	Use
SPN	Number of the sprite to be scrolled
NPX	Number of pixels to be scrolled

Command	Action
WCRM	Vertical scroll with wrap
SCRM	Vertical scroll, no wrap

Example:

To scroll sprite 5 downward by 11 pixels with wrap, use:

```
5 SPN ! -11 NPX ! WCRM
```

Attribute Scrolls

There are four commands to scroll attributes in any of the four directions:

Parameter	Use
SPN	Number of the sprite to be scrolled

Command	Action
ATTLM	Scroll attributes left with wrap
ATTRM	Scroll attributes right with wrap
ATTUPM	Scroll attributes up with wrap
ATTDNM	Scroll attributes down with wrap

GETS AND PUTS

There are three groups of GETs and PUTs. The first, and the fastest, carry out operations between a full sprite and a previously defined window of the screen. The second group carry out operations between sprite windows and screen windows. The third, and probably most powerful group of commands in the sub-language, cover operations between sprite windows and other sprites.

Suppose you have designed a sprite 160 characters wide (5 screens) and you wish to smoothly pixel scroll through this sprite via a screen window covering the bottom five character rows of the screen. A second dummy sprite, 1 character wider than the screen can be used to GET from the larger sprite, scroll, PUT, scroll, Put and so on for 1 character, then a second block, 1 character further into the larger sprite can be GOT and so on. Experimentation will soon show you how to do this. These routines can be run in background to provide fast smooth scrolling backdrops.

Group 1

This group provides block moves and logical operations between sprites and screen windows - these are the fastest commands in the set. This particular group does not include separate attribute commands but instead uses an attribute switch. If the switch is on, pixel data and attribute data are moved; if you do not require to move attributes you can increase the speed of operations by switching off the attribute switch.

Command	Action
ATTON	Enable the flow of attribute data between the sprite and the screen window
ATTOFF	Disable the flow of attributes between the sprite and the screen window

We now come to the group 1 commands themselves:

GETS

Parameter	Use
SPN	Number of the sprite to be used (1 to 255)
COL	Left hand column of target screen window (0 to 31)
ROW	Top row of target screen window (0 to 23)

Command	Action
GETBLS	Block move screen window into sprite
GETORS	OR screen window into sprite
GETXRS	XOR screen window into sprite
GETNDS	AND screen window into sprite

Note:

The dimensions of the screen window are taken as the dimensions of the sprite. If the command doesn't execute, it is almost certainly because the width of the sprite + COL or the height of the sprite + ROW, exceed 32 or 24 respectively, so that part of the window lies off the screen. Sprites must be previously set up either by the development software, or the COLD# and SPRITE or ISPRITE commands before these commands will execute.

Example:

To GET a screen sprite into sprite number 4 from column 5, row 4, with attributes and "OR" it with the data currently held in sprite number 4, use the following:

```
4 SPN ! 5 COL ! 4 ROW ! ATTON GETORS
```

Note that if ATTON was the last switch command, it would not be needed in the above.

PUTS

These commands are identical to the group 1 "GETS" except that data transfer is from the sprite to the screen. The results of the various operations are therefore displayed to the screen. The parameters are identical in operation to those of the "GETS", and the ATTON and ATTOFF switch commands also apply.

Parameter	Use
SPN	Number of the sprite to be used (1 to 255)
COL	Left hand column of target screen window (0 to 31)
ROW	Top row of target screen window (0 to 23)

Command	Action
PUTBLS	Block move sprite window into screen
PUTORS	OR sprite window into screen
PUTXRS	XOR sprite window into screen
PUTNDS	AND sprite window into screen

Group 2

These commands allow operations between sprite windows and screen windows. Unlike Group 1 commands, there are separate commands to move pixel data and attributes, and the ATTON, ATTOFF commands have no effect on their operation. Two new parameters are introduced to specify the COLUMN and ROW of the top left hand character of the sprite window. In addition, HGT and LEN are required to specify the dimensions of the window for the screen and sprite.

Parameters	Use
COL	Left hand column of target screen window (0 to 31)
ROW	Top row of target screen window (0 to 23)
SCOL	Left hand column of target sprite window (0 to sprite width -1)
SROR	Top row of target sprite window (0 to sprite height -1)
HGT	Height of window
LEN	Length of window
SPN	Sprite number

Command	Action
GWBLS	Get block of pixel data from screen window into sprite window
GWORS	OR pixel data from screen window into sprite window
GWXRS	XOR pixel data from screen window into sprite window
GWNDS	AND pixel data from screen window into sprite window
GWATTS	GET block of attribute data from screen window into sprite window
PWBLS	PUT block of pixel data from screen window into sprite window
PWORS	OR pixel data from sprite window into screen window
PWXRS	XOR pixel data from sprite window into screen window
PWNDS	AND pixel data from sprite window into screen window
PWATTS	PUT block of attribute data from sprite window into screen window

Note:

If the command does not execute, check that the window is not partially off the screen or sprite.

Example:

To block move a window 3 characters high and 4 characters wide from row 2, column 3 of the screen to row 4, column 6 of sprite number 7, use the following:

```
7 SPN ! 3 HGT ! 4 LEN ! 2 ROW ! 3 COL ! 4 SROW ! 6 SCOL ! GWBLS
```

Note that sprite 7 must be at least 8 characters wide and 7 characters high for the command to execute.

Group 3

This group, possibly the most useful in the whole set, comprises commands which support operations between sprites and sprite windows. The same set of commands as those in Group 2 are available and the format for each word is the same as in Group 2, except that the commands are postfixed with an "M" instead of an "S". The chief difference to the user lies in the set of parameters. The size of the data window is set to have the dimensions of the first sprite, and its position in the second sprite is set using the SCOL and SROW parameters.

Parameter	Use
SP1	Number of the first sprite
SP2	Number of the second sprite (containing the window)
SCOL	Left hand column of target sprite window
SROW	Top row of target sprite window

Command	Action
GWBLM	Block move pixel data from first sprite into window in second sprite
GWORM	OR pixel data from first sprite into window in second sprite
GWXRM	XOR pixel data from first sprite into window in second sprite
GWNDM	AND pixel data from first sprite into window in second sprite
GWATTM	Block move attribute data from first sprite into window in second sprite
PWBLM	Block move pixel data from window in second sprite into first sprite
PWORM	OR pixel data from window in second sprite into first sprite
PWXRM	XOR pixel data from window in second sprite into first sprite
PWNDM	AND pixel data from window in second sprite into first sprite
PWATTM	Block move attribute data from window in second sprite into first sprite

Note:

If the width of the first sprite added to SCOL, or the height of the first sprite added to SROW exceeds either the width or height of the second sprite respectively, then the command will not execute.

COPY COMMANDS

These five commands, are in fact a sub-group of the Group 3 commands and allow operations between pairs of sprites with the same dimensions. It is necessary, therefore, to provide only two parameters instead of four.

Parameter	Use
SP1	Number of first sprite
SP2	Number of second sprite

Command	Action
COPYM	Copy first sprite pixel data into second sprite
COPORM	OR first sprite pixel data into second sprite
COPXRM	XOR first sprite pixel data into second sprite
COPNDM	AND first sprite pixel data into second sprite
COPATTM	Copy first sprite attribute data into second sprite

Note:
If the dimensions of the two sprites are not identical, then the command will not execute.

Example:

To make a complete copy of sprite 10 into sprite 8, use the following:

```
8 SP1 ! 10 SP2 ! COPYM COPATTM
```

SPRITE TRANSFORMATIONS

To increase the utility of the package, four extra words have been added to invert, spin, reflect and enlarge sprites. The inversion and reflection routines work for screen and sprite data but the rotation and enlargement commands work only on sprites and a dummy sprite is required to rotate or enlarge into.

Inversions

The sprite is "1's complemented"; in other words, all the pixels which are set "on", became set "off" and vice-versa. The attributes, however, remain unchanged.

Parameter	Use
SPN	The number of the sprite to be inverted

Command	Action
INVM	The sprite is inverted (1's complemented)

Parameter	Use
COL	Column of the left hand edge of the screen window (0 to 31)
ROW	Row of the top edge of the window (0 to 23)
LEN	Width of the screen window (1 to 32)
HGT	Height of the screen window (1 to 24)

Command	Action
INVV	Invert screen window

Reflections

There are four commands in this group for reflecting screen and memory, pixel and attribute data. A sprite is often required to point in two directions and the command can either be used at the development stage or, if space is short, at run-time. The command causes horizontal reflection but vertical reflection is possible, by combining rotations and reflections.

Parameter	Use
SPN	Number of sprite to be reflected (1 to 255)

Command	Action
MIRM	Reflect sprite pixel data about its centre
MARM	Reflect sprite attribute data about its centre

Parameter	Use
COL	Column of left hand edge of screen window (0 to 31)
ROW	Row of top edge of the window (0 to 23)
LEN	Width of screen window (1 to 32)
HGT	Height of screen window (1 to 24)

Command	Action
MIRV	Reflect screen pixel data about window centre
MARV	Reflect screen attribute data about window centre

Spin

This command involves an operation between two sprites with transposed dimensions. If, for example, a sprite with dimensions 8 by 3 is to be spun into a second sprite, this second sprite must have dimensions 3 by 8. Square sprites are, of course, no problem. Pixel and attribute data are both rotated. If the command is to be used, it is important to remember that a second sprite will be needed to be rotated into and that it is necessary to set this up in advance. Rotation is 90 degrees clockwise.

Parameter	Use
SP2	Number of sprite to be rotated
SP1	Number of sprite to be rotated into

Command	Action
SPINM	Rotate 90 degrees clockwise sprite SP2 into sprite SP1.

Note:

A sprite cannot be rotated into itself, i.e. if SP1 and SP2 are the same number, the rotation will not work. The result is, however, well worth seeing! Data is "OR"ed from SP2 into SP1 so it is usually necessary to execute a CLSM to clear sprite SP1 before execution.

Enlargement

One command is provided for enlarging a sprite and its attributes into a second sprite which has dimensions exactly double those of the sprite being enlarged.

Parameter	Use
SP1	Number of the sprite into which the enlargement is carried out.
SP2	Number of the sprite being enlarged.

Command	Action
DSPM	Enlarge sprite SP2 into sprite SP1.

Note:

If the dimensions of sprite SP1 are not twice those of sprite SP2 the command will not execute.

INTERRUPT RELATED WORDS

Six interrupt related words are provided which control the Foreground/Background execution of White Lightning words. The first four have no parameters.

Command	Action
HALT	Suspend CPU operation until the next interrupt. Executing HALT in background mode will freeze the system permanently.
EI	Enable the interrupt.
DI	Disable the interrupt.
EXX	Exchange IDEAL variables with the alternate IDEAL variables. This command is executed automatically each time an interrupt occurs and at the end of the interrupt routine to restore foreground variables. If a background program is not being used EXX can be used to provide extra variables.
INT-ON	The specified word (see next example) is executed on the receipt of each interrupt.
INT-OFF	Following execution of INT-OFF the Z80 returns to interrupt mode 1 and polls the keyboard 50 times a second. The background program ceases execution.

Example:

To set a word running in background type:

```
' WORD INT-ON
```

Where "WORD" is the word to be run in background.

BASIC INTERFACE WORDS

These words are provided to enable you to mix BASIC and Forth in your program. The section covering the access of BASIC should be read carefully before an attempt is made to use these commands.

Command	Action
PROG	Enter BASIC at command level.
RESERVE	Reserve space in the dictionary for the insertion of BASIC source. The size of the space to be reserved is taken from the stack.

Example:

```
1024 RESERVE
```

will reserve 1k from the current value of HERE.

GOTO	The BASIC program is executed from the line number held on the stack. Forth can be called from BASIC using RANDOMISE USR 30000 and Forth can be re-entered using PRINT USR 30006.

Example:

```
100 GOTO
```

will commence execution of the BASIC source from line 100.

RETUSR	Control is returned to the BASIC program from Forth. Execution of the BASIC program proceeds with the first instruction after the RANDOMISE USR 30000 with which Forth was called. Do not execute a RETUSR in Forth if Forth has not been called using a RANDOMISE USR 30000 call.

MISCELLANEOUS WORDS

There are 18 words provided which cover general aspects of games development.

Command	Action
SETAV	Set the attributes to the current INK and PAPER colours in the screen window defined by HGT, LEN, COL, ROW.
SETAM	Set the attributes to the current INK and PAPER colours in the sprite whose number is held in SPN.
CLSV	Clear the screen window (defined by HGT, LEN, COL, ROW) of pixel data and set the attributes throughout to the current INK and PAPER colours.
CLSM	Clear the pixel data of the sprite whose number is held in SPN. Attribute data is unaffected.

ADJM This command is used to ensure the execution of group2 GET and PUT instructions. The parameters are SPN, COL and ROW. First HGT and LEN are set to the dimensions of the sprite whose number is held in SPN. If COL or ROW are "off screen" or if COL + LEN, HGT + ROW are off screen, then the parameters COL, ROW, HGT, LEN, SCOL, SROW are set such that the sprite will be partially PUT to the screen or GOT from the screen. This is an extremely useful command and can be used in conjunction with any of the group 2 GETs and PUTS.

Example:

```
-1 COL ! -1 ROW ! 1 SPN ! ADJM PWBLS
```

This will PUT part of sprite 1 in the top left hand corner. After execution, COL and ROW will have been made 0 and SCOL and SROW will have the value 1.

ADJV Essentially the same idea as ADJM but this time the screen window defined by HGT, LEN, COL, ROW is adjusted to lie "on screen"

SCANM The sprite whose number is held in SPN is scanned for pixel data. If data is found a true (non-zero) flag is placed on the stack, otherwise a false (zero) flag is placed on the stack. This command is used extensively for collision detection.

SCANV The character cell at screen positions defined by COL and ROW is scanned for screen data and a true or false flag stacked accordingly. This command executes more rapidly than SCAM.

RND Replace the value at the top of the stack by a random number between zero and the value at the top of the stack.

Example:

```
10 RND
```

will leave a number between 0 and 10 on the stack.

OUT# Output the second value on the stack to the port address at the top of the stack.

IN# Replace the port address an the top of the stack by the 16 bit representation of the 8 bit number read from the port.

Example:

The following will poll the Kempston Joystick and execute one of 8 words depending on the joystick position. Finally, the fire button will be tested. Type:

```
: JOYSTICK 31 IN# DUP CASE <CR>
 0 OF ." CENTRE " ENDOF <CR>
 1 OF ." RIGHT " FNDOF <CR>
 2 OF ." LEFT " ENDOF <CR>
 4 OF ." DOWN " ENDOF <CR>
 8 OF ." UP " ENDOF <CR>
 9 OF ." UPRIGHT " ENDOF <CR>
10 OF ." UPLEFT " ENDOF <CR>
 5 OF ." DOWNRIGHT " ENDOF <CR>
 6 OF ." DOWNLEFT " ENDOF <CR>
ENDCASE <CR>
15 > IF ." FIRE " ENDIF ; <CR>
: JTEST BEGIN JOYSTICK CR 1 1 KB UNTIL ;<CR>
```

To run type:

```
JTEST <CR>
```

To halt press CAPS SHIFT

KB This command is provided for the detection of multiple key presses. All it does, in fact, is test the specified key and stack a true flag if the key is pressed and a false flag if it is not. The key to be tested is specified by the top two numbers on the stack. The second value specifies the half ROW and the top value the COLUMN. For a full description of the COLUMNS and ROWS of the Spectrum keyboard see page 160 of the Spectrum manual. Below is a summary.

ROW	KEYS
1	CAPS SHIFT to V
2	A to G
3	Q to T
4	1 to 5
5	0 to 6
6	P to 7
7	ENTER to H
8	SPACE to B

Columns are organised from 1 to 5 and counted from the outside in. This is the order above.

BLEEP This operates in the same way as the Spectrum's BEEP command with the second number on the stack providing duration and top number pitch.

Example:

```
100 200 BLEEP
```

ATTON After the execution of ATTON, group 1 GETs and PUTs will GET and PUT attribute data at the same time as they GET and PUT pixel data. The GW and PW commands, however, are unaffected by ATTON or ATTOFF and always use separate commands to move pixel data. After the execution of a GW or PW command the attribute switch is always set to 'OFF'.

ATTOFF After the execution of ATOFF, group GETs and PUTs will move pixel data only.

CALL Control jumps to a machine code subroutine whose address is held at the top of the stack.

Example:

```
HEX D000 CALL
```

would execute a machine code subroutine at location D000 HEX. Don't type this unless you have a machine code routine at HEX D000!

ZAP Once program development is completed and you have compiled your final program into the dictionary, typing ZAP will produce a run-time version. The length of the final version is displayed to the screen and a copy can be saved by typing:

```
SAVE "filename" CODE 24832, LENGTH
```

the length being the length displayed. Typing:

```
PRINT USR 24832
```

will execute the last word defined before the ZAP command. This is the only form in which White Lightning programs can be commercially sold.

ZAPINT As above except that programs which utilise the Foreground/Background facility must be produced using the ZAPINT command as opposed to the ZAP command. The only real difference is that the first five screens also need to be saved and that the top 16 bytes of RAM will be used by the final program.

PRT-ON All subsequent screen output is to ZX-Printer only.

PRT-OFF All subsequent output is to screen only.

Example:

```
PRT-ON VLIST PRT-OFF
```

will list the current FORTH dictionary to the ZX-Printer.

FORTH/BASIC WORDS

For those users who have mastered and grown used to the Spectrum's own graphics commands, a set of 18 Forth implementations of Spectrum words is provided. Parameters are placed on the stack in the same order as they occur in their BASIC implementations. If an error occurs during the execution of a BASIC word, Forth should be re-entered via a WARM start, i.e. PRINT USR 24836. For a full description of the action of each of these words refer to your Spectrum manual.

PARAMETERS	WORD	ACTION
	COPY	The screen in 'dumped' to the ZX-Printer.
ROW, COL	**AT**	The print position is moved the specified Column and Row.
COLOUR	**BORDER**	The border colour is set to one of the 8 Spectrum colours.
	CLS	The screen is cleared of pixel data and the attributes set to the current INK, PAPER, BRIGHT and FLASH values.
X,Y,ANGLE	**DRAW-ARC**	See Spectrum manual page 122.
X,Y,RADIUS	**CIRCLE**	See Spectrum manual page 123.
X,Y	**DRAW**	See Spectrum manual page 121.
X,Y	**PLOT**	See Spectrum manual page 121.
ROW, COL	**SCREEN$**	The character at the screen position defined by the two values at the top of the stack is tested to see if it is one of the Spectrum's pre-defined characters. The ASCII value is left on the stack.
ROW, COL	**ATTR**	The code for the attribute at the screen position defined by the top two values on the stack is left on the stack.
X,Y	**POINT**	The pixel at the (x,y) co-ord defined by the top two values on the stack is tested and a true or false flag stacked depending on whether the pixel is set or not.
COL	**TAB**	Set the horizontal print position to the value at the top of the stack.
COLOUR	**INK**	Set the INK colour to the value at the top of the stack.
COLOUR	**PAPER**	Set the PAPER colour to the value at the top of the stack.
FLAG	**OVER.**	Note the full stop at the end of OVER. This sets the printing mode according to the value of FLAG which is zero or one.
FLAG	**INVERSE**	As for OVER.
FLAG	**BRIGHT**	As for OVER.

FOREGROUND BACKGROUND

Mastering machine code does give most programmers access to the speed of commercial games, but often the smoothness and continuity are lacking. One of the greatest difficulties facing any games designer is timing. The basic problem is that some parts of the program need to execute at regular intervals, and trying to achieve this can involve a lot of calculation and wasted processor time. The solution to this is to use interrupts to execute particular sections of code. White Lightning does this for you, using the words INT-ON and INT-OFF.

The Spectrum interrupt occurs 50 times a second, so background words can be executed at this frequency, or by counting interrupts, at lower frequencies.

If you list any of the screens 1 to 5, you will see that they are apparently filled with garbage. This is because the area in memory occupied by these screens contains the machine code that enables the background facility. If you are not intending to use this facility, then you can clear screens 1 to 5 and use them normally for source code. If you do this, however, don't forget that you won't be able to use any of the graphics words in background mode, or the system will crash in no uncertain manner!

When an interrupt occurs, the foreground program stops exactly where it is, saves off its parameters and then executes the background word. The background word will then execute fully before continuing execution of the foreground program, from the exact point at which it was halted. Three important points should be borne in mind. Firstly, if the execution time of the background word exceeds a fiftieth of a second, it is not possible to execute it more than twenty five times a second, if it exceeds a twenty-fifth of a second, it can only be executed at half that frequency, and so on. There is, however, no limit to the length of the background execution time itself. Secondly, as the execution time approaches a fiftieth of a second, or some multiple of a fiftieth of a second, then less and less processor time will be available for the foreground program and sometimes it is necessary to extend the length of the background program to make the foreground program run more quickly, by reducing the frequency of the background program. Experimentation will familiarise the user with the techniques required for the best effects. More foreground time can also be taken by disabling and then re-enabling the interrupt using DI and EI respectively. This brings us to the third, and most important point. Remember that when an interrupt occurs, the foreground program will stop whatever it is doing, execute the background program and then continue with the foreground execution. Suppose the background program is a sideways scroll of a user defined screen window and the foreground program PUTs a character into the window. A problem arises if an interrupt occurs halfway through the PUT, because the top half of the character will be scrolled before the second half of the character is PUT to the screen. To circumvent this problem, where an operation is carried out on the same screen or sprite data by both the foreground and background programs, the background program should be temporarily disabled using DI, the foreground word executed, and then the background program re-enabled using EI ready for the next interrupt to occur. The safest way to proceed until you have really mastered the language, is to avoid the situation altogether and make sure the foreground and background programs don't operate on the same sprite or screen area.

To set the background program running, simply type an apostrophe (shifted 7), a space, the word to be executed, a space and then INT-ON. Don't forget that if the background word does not set its own parameters, then these will need to be set before execution and if these are the IDEAL variables, then the alternate set will be used.

For example, suppose you wanted to scroll a window, four characters square, in the middle of the screen and invert it after each sideways scroll. First we need to define a word to do the scrolling and the inverting. For some reason, most test programs are called FRED and there is no reason for breaking with convention. To define the word type:

```
: FRED WRR1V INVV ; <CR>
```

To set up the parameters type:

```
4 HGT' ! 4 LEN' ! 14 COL' ! 9 ROW' ! <CR>
```

To make sure there is some data in the window, type:

```
VLIST <CR>
```

You are now ready to execute the background program by typing:

```
' FRED INT-ON <CR>
```

To halt this program type:

```
INT-OFF <CR>
```

This program is running a bit too fast to see, so let's write another program which slows this down to every fifth interrupt, i.e. ten times a second. We will need to define a variable and a new word. To set up the variable type:

```
0 VARIABLE ICNT <CR>
```

This sets up a variable called ICNT and assigns to it the value 0. We'll call this new background word FREDA. Type:

```
: FREDA ICNT @ 1+ 5 > IF FRED 0 ICNT ! ELSE 1 ICNT + ! ENDIF ; <CR>
```

Now type:

```
' FREDA INT-ON <CR>
```

All "FREDA" does, is to increment ICNT and compare it with 5 and if it is greater than 5 then "FREDA" is executed and ICNT set back to zero.

It would be useful to be able to control the speed that "FREDA" ran at, so, let's modify "FREDA" to do this. First, type INT-OFF FORGET FREDA <CR> to get rid of the old definition and then set up a new variable and construct a slightly different program. A variable which sets the limit on the number of interrupts needs to be set up, so type: 4 VARIABLE LCNT <CR>. The new definition is set up by typing:

```
: FREDA ICNT @ 1+ LCNT @ > IF FRED 0 ICNT ! ELSE 1 ICNT +! FNDIF ; <CR>
```

To execute the new word type:

```
0 ICNT ! ' FREDA INT-ON <CR>
```

This program increments ICNT, compares it with LCNT and executes when ICNT is equal to LCNT. Increasing LCNT then, will slow the background execution and decrementing LCNT will speed it up. If LCNT is put equal to 1, execution will occur every cycle.

To speed up "FREDA" type: 0 ICNT ! 2 LCNT ! <CR> and to slow "FREDA" type 10 LCNT ! <CR> and so on. Type: INT-OFF <CR> to halt FREDA. Now type: FORGET FREDA <CR>

Frequency and Phase

One of the problems with executing a word on each interrupt, is that the dot scanning the screen may overtake the screen operation in the same position on each execution. This can produce some strange effects, and often, sections of the screen window will appear to be "sliced". It is more usual to execute on selected interrupts. We can do this very simply using modular arithmetic.

Suppose we have four different words that we wish to execute with four different frequencies. Suppose they are as follows.

```
INVV   every 50 cycles
MIRV   every 20 cycles
WRR1V  every  4 cycles
WCRV   every  5 cycles
```

We now define a variable to count interrupts and four constants to store the frequencies.

```
0 VARIABLE ICNT 50 CONSTANT F1 20 CONSTANT F2 4 CONSTANT F3 5 CONSTANT F4 <CR>
: MAO MOD ABS 0= ; <CR>
: IRUN ICNT @ DUP DUP DUP <CR>
F1 MAO IF INVV ENDIF <CR>
F2 MAO IF MIRV ENDIF <CR>
F3 MAO IF WRR1V ENDIF <CR>
F4 MAO IF WCRV ENDIF <CR>
1 ICNT @ +! ; <CR>
```

All we need to do now is set up the parameters by typing:

```
10 COL' ! 10 ROW' ! 6 HGT' ! 8 LEN' ! 2 NPX' ! <CR>
```

then put some data on to the screen and execute:

```
VLIST 0 0 AT ' IRUN INT-ON <CR>
```

To terminate, type INT-OFF <CR>

If we had made the F1 to F4 variables, we could have controlled the background program from the foreground by resetting them.

Sometimes, controlling the frequencies of events is not sufficient and phase information needs to be introduced. In the previous example, values of 0, 100, 200, 300 and so on cause all four events. Supposing we wanted to maintain these frequencies, but change the order in which the words execute - we need to introduce the concept of phase.

In this example we need four more constants, so type:

```
FORGET IRUN <CR>

31 CONSTANT PH1 5 CONSTANT PH2 0 CONSTANT PH3 3 CONSTANT PH4 <CR>
```

Now type:

```
: IRUN ICNT @ DUP DUP DUP <CR>
F1 MOD ABS PH1 = IF INVV ENDIF <CR>
F2 MOD ABS PH2 = IF MIRV ENDIF <CR>
F3 MOD ABS PH3 = IF WRR1V ENDIF <CR>
F4 MOD ABS PH4 = IF WCRV ENDIF <CR>
1 ICNT +! ; <CR>
```

This can be executed using VLIST ' IRUN INT-ON <CR>
Halted using: INT-OFF <CR>
and cleared using: FORGET IRUN <CR>

Forth/BASIC Words

Sinclair's graphics and sound commands have been replicated as Forth words for continuity - they also execute slightly more rapidly than their BASIC counterparts. For a full list, see the section on Forth/BASIC WORDS.

They all execute code in Sinclair's own ROM which, unlike Forth and IDEAL, is not re-entrant. This means that these words cannot be executed in foreground and background simultaneously. The FORTH words ." , . and U. should also not be executed simultaneously with themselves or any of the BASIC words. None of the Forth/BASIC words, ." , . or U. should be executed in background while Forth is in command mode.

LOGICAL OPERATIONS

There are three types of logical operation included in the IDEAL sub-language; these are OR, XOR and AND. To get the best out of this package it is important to make full use of these commands.

If a GET or PUT postfixed with "BLS" or "BLM" is executed, then data is block moved from the source which may be part of the screen, a sprite, or a sprite window, in such a way that whatever was previously held at the destination which may also be part of the screen, a sprite, or a sprite window, is obliterated and replaced by whatever was at the source. This may not always be the desired effect and quite often the user will want to merge characters or remove parts of the characters and so on.

If two sprites are "OR"ed together, the resulting sprite will have pixels set where pixels were set in either or both of the sprites being "OR"ed.

If two sprites are "AND"ed together, the resulting sprite will have pixels set where pixels were set in both of the sprites being "AND"ed.

If two sprites are "XOR"ed together, the resulting sprite will have pixels set where pixels were set in either but reset where pixels were set or reset in both.

These results are summarised as follows and should make things a little clearer:

SOURCE	**DESTINATION**	**OPERATION**	**RESULT**
on	on	OR	on
on	off	OR	on
off	on	OR	on
off	off	OR	off
on	on	OR	on
on	off	OR	off
off	on	OR	off
off	off	OR	off
on	on	XOR	off
on	off	XOR	on
off	on	XOR	on
off	off	XOR	off

We can now use the sample sprites to illustrate the effects of these operations. First FORGET any previously defined words and type: DECIMAL <CR> to ensure that you are in decimal mode. Now type:

```
7 INK 0 PAPER CLS ." LOGICAL OR " CR ." LOGICAL OR " <CR>
```

This will clear the screen and put some data in the top left hand corner.

Now type:

```
3 SPN ! 0 COL ! 0 ROW 1 ATTOFF PUTORS <CR>
```

You will see that the data has been merged together, both the dragster and the letters remain.

Now type:

```
CLS ." LOGICAL AND " CR ." LOGICAL AND " PUTNDS <CR>
```

This time, the only data remaining is at those points where the data coincided. Logical "AND"s are normally used to mark off sections of the screen or sprites. They are also used extensively for collision detection. If a window of the screen is "AND"ed into a sprite, and then a SCANM performed, it is possible to determine whether a collision would occur if the sprite were PUT; before actually PUTting the sprite.

We now come to the logical XOR. This is probably the most useful operation of the lot. "XOR"s have the peculiar property of restoring the destination data to its former state if the operation is performed twice. This is how Sinclair's own "OVER" operation works. To see this happen type:

```
CLS ." XOR " CR ." XOR " PUTXRS <CR>
```

To restore the text type:

```
PUTXRS <CR>
```

We can even use this property to swap two sprites without using a third. This example will swap the data but not the attributes. We will "PUT" two sprites at each stage so you can see what is happening. Type:

```
CLS ATTON 0 SCOL ! 0 SROW ! 1 SP1 ! 2 SP2 ! 23 COL ! 6 0 AT <CR>
```

Now type:

```
1 SPN ! PUTBLS 2 SPN ! 27 COL ! PUTBLS <CR>
```

These are the two sprites before the operations begin.

Now type:

```
GWXRM 2 ROW ! 23 COL ! 1 SPN ! PUTBLS 27 COL ! 2 SPN ! PUTBLS <CR>
```

sprite 1 has now been "XOR"ed into sprite 2. Now type:

```
PWXRM 4 ROW ! 23 COL ! 1 SPN ! PUTBLS 27 COL 1 2 SPN ! PUTBLS <CR>
```

sprite 2 is now in sprite 1. Finally, type:

```
GWXRM 6 ROW ! 23 COL ! 1 SPN ! PUTBLS 27 COL ! 2 SPN ! PUTBLS <CR>
```

The operation is now complete.

If we wanted to add a new word to the language which swaps two equally sized sprites whose numbers were in SP1 and SP2, we would now do so.

```
: SWOP 0 SCOL ! 0 SROW 1 GWXRM PWXRM GWXRM ; <CR>
```

In fact we could also use:

```
: SWOP 0 SCOL ! 0 SROW ! PWXRM GXXRM PWXRM ; <CR>
```

for exactly the same effect. You will get MSG # 4 if you type the second word before typing FORGOT SWOP <CR>

Ironically, even though we can swap pixel data, there is no simple method for swapping attribute data unless a third sprite is involved.

COLLISION DETECTION AND SPRITE RECOGNITION

Two words are provided for collision detection, these are SCANV and SCANM.

SCANV is used to scan a particular character position on the screen. If any data is present in the specified square (co-ords are held in COL and ROW), then a true flag is placed on the stack and if the square is empty (contains no pixel data), a false flag is placed on the stack. Type:

```
CLS VLIST 0 VARIABLE CNT <CR>
```

This will put some data on the screen and initiate the variable CNT. Now type:

```
: GO 0 CNT ! 24 0 DO I ROW ! 32 0 <CR>
DO I COL ! SCANV IF 1 CNT +! <CR>
ENDIF LOOP LOOP CNT ? ; <CR>
```

This defines a word which simply counts the number of characters on the screen. Type:

```
GO <CR>
```

This should print a number somewhere around 250.

Often it is insufficient to determine whether a particular character square contains data or not, and for this reason the slower, but more powerful command SCANM, has been included. This will scan the sprite whose number is held in SPN and put a true flag on the stack if the sprite contains pixel data, or a false flag if it does not. SCANM is normally used to perform one of three functions:

1. To see if data will collide.

2. To detect an exact pattern.

3. To detect the presence of a pattern.

Collision detection is most commonly used to detect a collision between a sprite moving across the screen and any data which lies in its path. Often the sprite can pass through an occupied character position without a collision occurring, so the SCANM command is insufficient. The procedure is basically to load a dummy sprite with the section of screen into which the sprite is about to be put, "AND" it with the sprite about to be PUT and then use SCANM. If a true flag is on the stack the dummy sprite contains data and therefore a collision has occurred. This is all very well, but a problem occurs if the new sprite position overlaps the old sprite position, because this means that the old sprite has to be removed from the screen before beginning the above detection procedure and subsequently PUTting the new sprite. This delay causes flicker. The easiest solution is to work with "XOR"s so that the window can be GOT, "XOR"ed with the old sprite in memory to remove the old sprite data, and then to do the detection followed finally by the blotting and then immediate PUTting.

Once an impending collision is detected it is frequently useful to determine what the sprite has collided with. To begin with, let's assume that the screen window we're examining contains one of a known set of objects and that no other data is present in the window. The method is to load the dummy sprite with the object to be tested and then compare it against the set of sprites with which a match is being sought. To compare the dummy sprite with a known sprite, all you need to do is XOR the sprite being tested into the dummy and do a SCANM. If the result is zero, an exact match was found, if not, do a second XOR into the dummy to restore it and test the next candidate.

Finally, consider the case where the object being tested contains extraneous data in addition to one of the possible sprites. This time, the dummy sprite is loaded with the contents of the screen window, but the candidates are first "AND"ed into the dummy to remove extraneous data before the XOR and SCANM. Finally the dummy needs to be reloaded from the screen before the next test. This latter test is limited by the fact that its conclusion is only that the screen contained all the parts of the sprite with which a comparison was made. In the extreme case of the screen window containing all pixels set, then an agreement would be found with all the sprites tested.

SCROLLING LANDSCAPES

Scrolling landscapes are an integral part of so many video games that it is worth a brief description of how they can best be produced using White Lightning.

The first and most obvious point is never to scroll more than you have to. If, for instance, you are moving a mountain range where the variation takes place over the top three characters, then only the top three characters need to be stored and moved.

The simplest and most effective method of producing smooth scrolls is to sacrifice a column of the screen for transactions with the sprite being scrolled. Suppose we are scrolling a sprite of 4 or 5 screens width which uses rows 8 to 10 (3 rows). Suppose we require pixel scrolling and there is no horizontal variation in attributes. It doesn't really matter which column we sacrifice, far right (column 31) or far left (column 0), but let's, for this example, use column 0. All that we need to do is set up a window 1 character wide and 3 characters high on the far left of the landscape to have the same INK and PAPER colours. This means that pixel data cannot be seen in this region. Use the SETAV command to do this. To begin with, 31 columns of the sprite are PUT to the active part of the screen using the PWBLS command. If scrolling is to the left, then the dummy column should be loaded with the next column to the right of the sprite now 'on screen'. If scrolling is to the right then the column to the left of the sprite window should be inserted. The full 32 column screen window is now wrapped in the appropriate direction until a total of + or - 8 pixels has been accrued. The dummy column is then loaded from the appropriate sprite column and so on. The method can be simply adapted to make the landscape wrap and is usually implemented under interrupt.

PROGRAMMABLE SPRITES

One of the most common applications of the background mode is the setting of sprites into automatic motion. Perhaps the chief advantage that a language has over a games designer, is that the sprites thus created can have as much 'intelligence' as the programmer requires. A sprite can bounce off the edge of the screen and/or other sprites until a particular event, and then totally change its behaviour - possibly to follow a previously stored track.

We have included a very simple listing which sets a sprite in motion, that just bounces off the edges of the screen, to give you some idea of what is involved. This sample program assumes you have the demonstration sprites in memory.

```
SCR # 6
  0
  1
  2
  3 0 VARIABLE DELAY : WAIT DELAY @ 0 DO NOOP LOOP ;
  4 : BASE 0 COL ! 18 ROW ! 32 LEN ! 6 HGT ! 0 PAPER 0 INK
  5 SETAV 7 INK ;
  6 : GO16 6 PAPER 0 INK 0 BORDER CLS 7 0 AT
  7 10 0 DO ." WHITE LIGHTNING " LOOP 0 0 AT BASE ; -->

SCR # 7
  0 8 VARIABLE PX 8 VARIABLE PY 1 VARIABLE DX 1 VARIABLE DY
  1 0 VARIABLE SP 0 VARIABLE CL 0 VARIABLE RW : PCAL PX @ ABS
  2 2 /MOD CL ! PY @ ABS 2 /MOD RW ! DUP + + 251 + SP ! ;
  3 : MOVE PX @ 56 > IF DX @ MINUS DX ! ENDIF PX @ 0 > IF NOOP
  4 ELSE DX @ MINUS DX ! ENDIF
  5 PY @ 28 > IF DY @ MINUS DY ! FNDIF PY @ 0 > IF NOOP ELSE DY @
  6 MINUS DY ! ENDIF DY @ PY @ + PY ! DX @ PX @ + PX ! ;
  7 : LO RW @ ROW ! CL @ COL 1 SP @ SPN ! ; -->

SCR # 8
  0 : SOT PCAL LD EXX LD EXX PUTXRS ;
  1 : GO PCAL MOVE PUTXRS LD PUTXRS ;
  2 0 VARIABLE ICNT 2 VARIABLE LCNT
  3 : IRUN 1 ICNT +! ICNT @ DUP 2000 = IF -2 DX ! ENDIF DUP 4002 =
  4 IF 2 DY ! ENDIF DUP 6000 = IF 3 DY ! ENDIF DUP 8000 = IF 1 DY
  5 ! 1 DX ! ENDIF 9000 = IF INT-OFF ENDIF ;
  6 : TRY SOT ' GO INT-ON 9000 0 DO IRUN LOOP  INT-OFF ;
  7 : SCN16 GO16 9999 DELAY ! WAIT 1 DX ! 1 DY ! 8 PX ! 8 PY ! -->

SCR # 9
  0 0 ICNT ! TRY WAIT 0 PAPER 7 INK CLS ;
  1
  2
  3
  4
  5
  6
  7
```

To compile this type:

```
6 LOAD <CR>
```

and to execute type:

```
SCN16 <CR>
```

THE BASIC INTERFACE

The BASIC interface was provided to increase the flexibility of the language and allow the newcomer to Forth, a gradual transition. Some applications are actually more suited to BASIC but for games writing in general, Forth is much more appropriate and we hope that this facility will not discourage people from experimenting with Forth.

There are four words to master at the Forth end and 3 USR calls to master at the BASIC end. Do not use CLEAR or NEW whilst in BASIC.

The Command Level

When White Lightning is first entered from a COLD start, BASIC is located beneath Forth and there is approximately 1k of program space if microdrives are not in use. This is ample space if BASIC is only to be used at command level, to LOAD and SAVE for instance, but if programs are to be written you will need to execute the RESERVE command. For the time being, however, let's just consider operation at the command level. To enter BASIC from Forth type:

```
PROG <CR>
```

To re-enter Forth from BASIC just use:

```
PRINT USR 24836 <CR>
```

This is the normal WARM start entry. Note that PRINT USR must be used and not RANDOMIZE USR, or an OUT OF SCREEN error may occur.

BASIC AS A SUBROUTINE

At the next level, lines of BASIC can be executed as if they were subroutines and then return made to your Forth program. The word at the Forth end is GOTO. To return to Forth and continue execution use PRINT USR 30006.

To begin with, space needs to be made in the dictionary for the basic program. The word used to do this is RESERVE. What RESERVE actually does, is to make space in the dictionary and reset BASIC's system variables to point to this new area. This does mean, however, that if a second reserve is done, without FORGETting the old space, then the old space is lost and can never be re-accessed. Do not execute a Forth COLD start while BASIC is reserved or a RAMTOP error may occur if insufficient memory is reserved. Always execute PROG as the next command after RESERVE.

As an example, try the following:

```
DECIMAL 2000 RESERVE PROG <CR>
```

This will set up the BASIC space and then enter it at command mode. The following lines of BASIC can now be entered:

```
1000 PRINT "LINE 1000 OF BASIC" : PRINT USR 30006
2000 PRINT "LINE 2000 OF BASIC"
2010 FOR I = 1 TO 8 : PRINT I : NEXT I
2020 PRINT USR 30006
```

After entering these lines type:

```
PRINT USR 24836
```

to re-enter Forth at command level.

Let's now define a word which executes some Forth, some BASIC, some more Forth, some more BASIC, and finally some more Forth. To begin executing BASIC at a particular line, all that we need to do is put the line number on the stack and then execute GOTO. Try the following:

```
: FBDEM ." IN FORTH " CR 1000 GOTO ." BACK IN FORTH " <CR>
CR 2000 GOTO ." FORTH AGAIN " ; <CR>
```

Now type FBDEM <CR>

A more useful application would be to define words to handle cassette loading and saving. BASIC source is saved and loaded in the normal way from the reserved BASIC area.

Forth As A Subroutine

If you're an "out and out BASIC person" you're probably more likely to want to execute Forth as a subroutine. To return to BASIC from Forth use the word RETUSR. To call Forth from BASIC use RANDOMIZE USR 30000. Note that on this occasion it is a RANDOMIZE USR and not a PRINT USR. Using the previously reserved space we can try another example. First type:

```
PROG <CR>
```

to enter BASIC, then add the following lines:

```
3000 PRINT " CALLING FORTH " : RANDOMIZE USR 30000
3010 PRINT " BACK IN BASIC " : PRINT USR 30006
```

Now type PRINT USR 24836 to re-enter Forth.

Now type:

```
: BFDEM ." GOTO BASIC " CR 30000 GOTO ." FORTH CALLED " <CR>
RETUSR ." ENDING IN FORTH " CR ; <CR>
```

Now type

```
BFDEM <CR>
```

to see the result.

Now type

```
FORGET FBDEM <CR>
```

Passing Parameters

Forth variables can easily be PEEK'ed and POKE'd from BASIC and used not only to pass data, but also to control the execution of Forth. As an example, suppose we wished to select one of 4 Forth words at any one time with a call from BASIC. Let the Forth words simply be ." WORD1 ", ." WORD2 ", ." WORD3 ", ." WORD4 ". First we'll need a variable to pass the parameter, so type:

```
0 VARIABLE CONTROL CONTROL U. <CR>
```

This will set up a variable called CONTROL, set it to zero and then print the address of the least significant byte which we'll use to pass the information. For the sake of this example suppose the address was 50000. We'll now use the CASE construct to select the word to execute. Use the following definitions:

```
: SELECT CASE 1 OF ." WORD1 " ENDOF 2 OF ." WORD2 " <CR>
ENDOF 3 OF ." WORD3 " ENDOF 4 OF ." WORD4 " ENDOF <CR>
ENDCASE CR ; <CR>
```

If the value in CONTROL is 1 to 4, the appropriate word will be executed.

```
: RUN 4000 GOTO BEGIN CONTROL @ <CR>
DUP SELECT DUP IF RETUSR ENDIF 0= UNTIL ; <CR>
```

The BASIC program is initially entered at 4000 and could take the following form:

```
4000 REM REPLACE ADDRESS 5000 WITH THE ADDRESS OF CONTROL
4010 PRINT " EXECUTE WORD1 " : GOSUB 5000
4020 PRINT " EXECUTE WORD2 " : GOSUB 5010
4030 PRINT " EXECUTE WORD3 " : GOSUB 5020
4040 PRINT " EXECUTE WORD4 " : GOSUB 5030
4050 PRINT " FINISH " : POKE 50000,0 : PRINT USR 30006
5000 POKE 5000,1 : RANDOMISE USR 30000 : RETURN
5010 POKE 5000,2 : RANDOMISE USR 30000 : RETURN
5020 POKE 5000,3 : RANDOMISE USR 30000 : RETURN
5030 POKE 5000,4 : RANDOMISE USR 30000 : RETURN
```

Note that when final return is made to Forth a PRINT USR 30006 is used. If a RANDOMISE USR 30006 CALL is made to Forth a RETUSR must be executed or the BASIC stack will be left corrupted. To reset the stack if it has been corrupted, use PROG to enter BASIC and then re-enter Forth with the WARM start, PRINT USR 24836.

This concludes the section on the BASIC Interface.

PROGRAM DEVELOPMENT

At any one time, there are up to five areas of development: Forth source code, BASIC source code, sprites, the Forth language itself, and finally the compiled and completed program.

Forth Source

As previously discussed under the section on editing, Forth source is divided into screens, each of 512 bytes in length. Each screen can be individually loaded, saved and compiled in any order required. Screens can even be saved and then loaded back into different screens. The real advantage of this comes when you're writing really large programs. As sprite space becomes large, it will work down over the higher screens and this can be clearly seen when an attempt is made to List them. Don't CLEAR these screens or the sprite data will be lost!

If really large programs are required and sprites have over-run the top screens, then programs can be compiled a few screens at a time, loading each time into the available screens, compiling and then loading the next section. Of course, you don't need to load the sprites until compilation is complete, but it's useful to have the facility just in case.

To save Forth source you'll need to consult Table 1, the table of Screen Addresses. If, for instance, you wanted to save screens 6 to 11, then the start address would be 52224 decimal and the length, just 6 times 512. Type 6 512 * . <CR> to find this figure, which is 3072. To save the source, type PROG to enter BASIC and then type:

```
SAVE "filename" CODE 52224,3072
```

To re-enter White Lightning, type PRINT USR 24836 to do a WARM start.

To Load the source, either type Y in response to the LOAD SOURCE Y/N prompt at the beginning of the session, or exit to BASIC using PROG, then type:

```
LOAD "filename" CODE     where filename is optional.
```

If you want to load the code into a different screen area from that in which it was Saved, type:

```
LOAD "filename" CODE start, length
```

where start is the address of the screen to be loaded, and length is the number of screens to be loaded multiplied by 512. Again, White Lightning should be re-entered with a PRINT USR 24836. Do not use RANDOMISE USR 24836 or an OUT OF SCREEN error may occur.

BASIC Source

Before BASIC source can be used in White Lightning programs, the user must execute a RESERVE to make space for the BASIC program. To reserve, for example, 1K, type: DECIMAL 1024 RESERVE. This will allocate 1024 bytes for BASIC source code within the dictionary. If at some later stage you execute a second RESERVE the previous 1024 bytes are not reclaimed, so if you find you have not allocated enough space, Save the BASIC source, FORGET all previous definitions, execute a COLD START, and start the compilation from scratch. You can now do a second RESERVE.

To save BASIC source, type PROG <CR> to enter BASIC if you're not already there, then just type SAVE "filename" as normal and re-enter White Lightning with PRINT USR 24836. Likewise, source can be reloaded by entering BASIC with a PROG, using LOAD "filename" and then re-entering Forth with a PRINT USR 24836.

Sprites

Sprites can be saved from White Lightning and then re-Loaded into White Lightning, but sprites saved by White Lightning, cannot be loaded into the sprite development software which requires the additional array information preceding sprites which is not SAVEd by White Lightning. Sprite development should always be done using the development software, but if you do wish to save the sprites for later merging then do the following:

1. Find the start of sprites by typing SPST @ U.

2. Find the length to save by typing SPND @ SPST @ - 1+ U.

3. Note the start and length, then return to BASIC using PROG.

4. Save using SAVE "filename" CODE start, length.

5. Re-enter White Lightning using PRINT USR 24836.

Merging Sprites

Two blocks of sprites can be merged together in the main program using the following procedure:

1. Make a note of the SPST and SPND values of the second block to be merged. These are displayed by the sprite development software.

2. Load the main White Lightning package and then load the first block of sprites in response to the "LOAD SPRITES Y/N" prompt.

3. Load source as required and once in the main program relocate the first block of sprites downwards by the size of the second block. Suppose the decimal values for SPST and SPND of the second block were 60000 and 65280 respectively, then type:

   ```
   DECIMAL 60000 65280 - SPST @ + U. <CR>
   ```

   (The DECIMAL is not required if you are already in DECIMAL mode). This will calculate the new start after relocation. It is well worth checking that this will not run over your source code, so here is a quick calculation that will tell you if you have enough space. You need to know the highest screen number that you intend to use, for example screen 18. Type:

   ```
   18 512 * 49664 + U. <CR>
   ```

   This will print the first free byte after screen 18. So long as this result is lower than the new sprite start after relocation you can proceed. Again, using the previous example where the block to be merged has SPST and SPND of 60000 and 65280 respectively, the line to type is:

   ```
   DECIMAL 60000 65280 - MLEN ! RELOCATE <CR>
   ```

The relocate command uses the value held in MLEN as the relocation length, a negative value, as above, relocates downward and a positive value upward.

4. Before loading the second block of sprites, the values of the new SPST and SPND should be calculated and noted. Type:

   ```
   SPST @ U. SPND @ 65280 60000 - + DUP SPND ! U. <CR>
   ```

   Take a note of these two values. If the previous steps have been carried out correctly the second number (the new SPND) should be the same value as the old SPND before relocation.

5. Type: PROG <CR> to exit back to BASIC then type LOAD "" CODE. The array of pointers will be ignored but the sprites will be loaded. This assumes that this second block of sprites was also saved using the sprite development software.

6. Type PRINT USR 24836 to re-enter Forth and your sprites should be merged. Note that if a sprite number used in the second block has also been used in the first block, that only the first occurrence will be found. If the first occurrence is destroyed using WIPE or DSPRITE, then the second occurrence will be found.

Extending the Forth Itself

One of the beauties of the Forth language is that it is extendable, so if you've added a few of your own commands which you would like to become a permanent feature of your customised version, you will need to make a copy. For this reason, no attempt has been made to protect the software; but we do appeal to users not to take advantage of this facility to pirate the program. Piracy pushes up the price of software to genuine users, so if you've bought a genuine copy, do yourself a favour and keep the price of your future software affordable. Copying the manual, however, will result in immediate court action and a reward will be paid to anyone offering information leading to the successful prosecution of offenders.

To save the Forth use the following procedure:

1. Type: WARM->COLD <CR> to embed the commands.

2. Type: HERE 24832 - 1+ U. <CR> to print the length to be Saved.

3. Type: PROG <CR> to enter BASIC.

4. Save using SAVE "FORTH" CODE 24832, length.

5. Re-enter using PRINT USR 24836.

To use the amended version, LOAD White Lightning as normal, exit using PROG, LOAD the new Forth over the old Forth and execute a Cold start using 24832.

Oasis make no undertakings to support customised versions, and make no guarantee as to the success of the operation.

Compiled and Completed Programs

Once the program is fully debugged and running, a final run-time version can be produced. This is the only form in which programs generated from White Lightning can be marketed.

If the program makes use of the foreground/background facility, ZAPINT should be typed, if not, then ZAP should be typed. The length of the compiled program is then displayed until a key is pressed and control returned to BASIC to make a copy.

The final program should be saved using:

```
SAVE "filename" CODE 24832, length
```

and executed using PRINT USR 24832. Do not use RANDOMISE USR 24832.

Remember that a lot of run-time software is saved with your final code, so even if your program is only two lines long, the resulting program will be pretty large.

TABLE 1
Table of Screen Numbers and Addresses

Screen Number	Start Address
1	49664
2	50176
3	50688
4	51200
5	51712
6	52224
7	52736
8	53248
9	53760
10	54272
11	54784
12	55296
13	55808
14	56320
15	56832
16	57344
17	57856
18	58368
19	58880
20	59392
21	59904
22	60416

Each screen used for editing into consists of 8 lines x 64 characters = 512 bytes.

Therefore, if you have only edited into screens 6-9, then there is no need to save ALL of the screens 1-22 since you only need save from 52224 to 54271 (end of screen 9), i.e. 2k bytes.

FUNCTION KEY SUMMARY

KEY

A Activates the ATIRIBUTE switch.
Press 1 to set switch ON.
Press 0 to set switch OFF.

B Activates the BRIGHT variable.
Press 1 to set BRIGHT to ON.
Press 0 to set BRIGHT to OFF.

C Activates the PAPER variable.
Press any key between 0 and 7 to activate the colour indicated above the key.

D Activates DIRECT DATA INPUT
Accepts 8 bytes of data, one byte at a time, followed by ENTER, via the keyboard, to the position on the sprite Screen indicated by the cursors. Inputted data must be in the range 0 to 255 Decimal or, H00 to HFF HEX (the character H must precede Hex entry).

NOTE: If Attribute switch = 1, then the four current attributes will be used at the same position as well.

E Activates the SCREEN FUNCTIONS.
You will be given three options: press 1, 2 or 3.

1 INVERT
Option 1, INVERT, sets all 0 bits to 1 and all 1 bits to 0, in a window whose length is held in the "Sprite length" variable and whose height is held in the "Sprite height" variable. The inversion will take place from the positioning of the sprite screen cursors, i.e. at the intersection of an imaginary line drawn from each cursor.

2 MIRROR
Option 2 MIRROR, 'Flips' a window whose height is held in the "Sprite height" variable and whose length is held in the "Sprite length" variable. The Mirroring will take place about the vertical centre of the screen window.

3 MIRROR ATTRIBUTES
Option 3 MIRROR ATTRIBUTES, 'Flips' the attributes in a window whose height is held in the "Sprite height" variable and whose length is held in the "Sprite length" variable. The Mirroring of Attributes will take place about the vertical centre of the screen window

F Activates FLASH WINDOW.
Flashes the current screen window whose height is held in the SPRITE HEIGHT variable and whose length is held in the SPRITE LENGTH variable. The Flash will take place at the position of the sprite screen cursors.

Flash is used to check the position of the sprite screen cursors, to check that the height and length parameters are as required or to check that the window is correctly positioned.

G Activates GET SPRITE function.
Gets a sprite of the dimensions held in the "Sprite height" and "Sprite length" variables, using the number held in the "Sprite Number" variable and at the window indicated by the sprite screen cursor - and stores it in memory.

NOTE: If the Attribute switch = 1, the sprite and attributes are stored; if the Attribute switch = 0, then any Attributes will be ignored. When a sprite is first defined with Attribute switch = 0 the attribute data will probably be garbage.

H Activates the SPRITE HEIGHT Variable.
Permits the input of the height of a sprite window from 1-15.

I Activates the ATTRIBUTE DUMP function.
Places the four Attributes set in the four Attribute Variables, to the sprite screen, at the position indicated by the sprite screen cursors - with a resolution of one character.

NOTE: This function is independent of the Attribute Switch, e.g. to place Attributes at position x=4, y=4: position sprite screen cursors at x=4 and y=4, then set the four attributes as required (you can then set the Attribute switch to 0 (OFF) if you like) and press I.

J Activates the move CHR$ SQR TO SPRITE SCREEN function.
Dumps the bit pattern set in the CHR$ SQR to a character square in the sprite screen, indicated by the sprite screen cursors.

NOTE: If the Attribute Switch = 0, no Attributes will move with the pattern. If the Attribute switch = 1, then the Attributes held in the Attribute Variables will move with the pattern.

K Activates the MOVE SPRITE SCREEN CHARACTER TO CHR$ SQR function.
Picks up the Character Square indicated by the Sprite Screen Cursors, into the CHR$ SQR.

NOTE: ATTR = 0 ignores Character Attributes. ATTR = 1 takes the Attributes of the character and loads them into the Attribute Variables.

L Activates the SPRITE LENGTH variable.
Permits the input of the length of a Sprite Window from 1-15.

M Activates the Sprite Functions.
You will be given three options which act in the same way as the 'SCREEN FUNCTIONS E', except that these functions operate on the sprite in memory only and have no effect directly on the screen.

N Activates the No, negative response to (Y/N) questions.

O Activates the Sprite Logic functions.
You will be given three options. Each option GETS an area of the sprite screen, the dimensions of which are specified as those of the defined sprite, having a top left-hand corner at the sprite screen cursor positions and logically GETs the data into the defined sprite - whose number is in the Sprite Number Variable.

NOTE: ATTR = 0 leaves the attributes of the sprite as they are. ATTR = 1 takes the attributes from the screen and places them into the sprite.

1 GETORS, ORs the screen data with the pre-defined sprite, and leaves the result in the sprite (screen display unaffected).

2 GETXRS, XORs the screen data with the data of a pre-defined sprite, and leaves the result in the sprite, (screen display unaffected).

3 GETNDS, ANDs the screen data with the data of a pre-defined sprite, and leaves the result in the sprite (screen display unaffected).

P	Activates the PUT functions. You will be given four options. Each option PUTS a sprite whose number is specified in the variable "Sprite Number" onto the sprite screen, having a top left-hand corner at the sprite screen cursor positions.

NOTE: ATTR = 0 leaves the Screen Attributes unaffected. ATTR = 1 PUTs sprite Attributes to the sprite Screen.

1 PUTBLS is a straightforward PUT, placing data directly to the sprite screen, destroying anything that is on that part of the screen (Sprite unaffected).

2 PUTORS, ORs the sprite data with the data on the sprite screen, leaving the result on the screen (Sprite unaffected).

3 PUTXRS, XORs the sprite data with the data on the sprite screen, leaving the result on the screen (Sprite unaffected).

4 PUTNDS, ANDs the sprite data with the data on the sprite screen, leaving the result on the screen (Sprite unaffected).

Q	Activates the CLEAR CHR$ SQR function. Sets all CHR$ SQR bits to zero.
<= (SYMBOL SHIFT Q)	Activates the CLEAR SPRITE SCREEN function. Clears the sprite screen of all data and attributes.
R	Activates the ROTATE SPRITE function. Rotates a sprite, in memory, by 90 degrees, leaving the original sprite unaffected. The new Rotated sprite must be given a new sprite number, as asked for. Attributes are automatically Rotated with the pixel data.
S	Activates the SPRITE NUMBER variable. Permits the defining of sprites and asks for a sprite number in the range 1 to 255

NOTE: If a sprite to be defined is given an existing sprite number, a warning is displayed, advising you of this fact. The existing sprite, or the new sprite, are in no way corrupted.

T Activates the TEST SPRITE function.
Performs a test on the sprite whose number is held in the "Sprite Number" variable, and does the following:

1. Places the sprite height into the "Sprite height" variable.
2. Places the sprite length into the "Sprite length" variable.
3. Places the address in memory of where the sprite data starts, into the "Sprite" variable.
4. Places the address of the start of sprite space into the variable "SPST"
5. Places the address of the end of sprite space into the variable "SPND".
6. Calculates the remaining memory available for sprite storage and places it into the "Memory Left" variable.

NOTE: The screen display of these variables will be updated if necessary.

U Activates the PICK UP ATTRIBUTES function.
Picks up the attributes of the character from the sprite screen, indicated by the position of the sprite screen cursors and Loads them into the four Attribute variables.

V Activates the FLASH variable. This is one of the four attributes.
Press 1 to put switch ON.
Press 0 to put switch OFF.

W Activates the WIPE SPRITE function.
Wipes the sprite indicated by the "Sprite number" variable totally from memory. All other sprites stored in memory below that sprite are moved up to fill the space previously occupied by the Wiped sprite.

X Activate the INK variable which is one of the four attributes.
Press any key between 0 and 7 to set the colour indicated above the key.

Y Activates the YES, positive response to (Y/N) questions.

Z Activates the pre-defined ARCADE CHARACTER function.
Place a pre-defined Arcade Character to the sprite screen.
The top left hand corner of the character is indicated by the sprite screen cursors. Input a number between 1 and 167 followed by ENTER.

NOTE: Each character, with its number, can be seen on the Demo B tape. A list is given at the back of this section.

NOT (SYMBOL SHIFT S) Activates the SAVE SPRITES TO TAPE facility.
Place a suitable cassette in your cassette recorder and position as desired. Press NOT, enter your filename (1 to 8 characters). The program will save three groups of data; an array and two sections of code.
After SAVEing, you will be asked to rewind the tape and VERIFY - be sure to only press PLAY on your cassette recorder.
If the programs VERIFY, the Sprite Development Program will return to command level with the Text Line cleared.

NOTE: If the program breaks because of failure to VERIFY, type GOTO 3 and execute a WARM START; your data will not be lost.

- (SYMBOL SHIFT J)	Activates the LOAD SPRITES FROM TAPE facility. Place tape in your cassette recorder. Press SYMBOL SHIFT J and press PLAY on the cassette recorder. Three groups of data will load. When loaded, the Text Line will clear and the program will resume. NOTE: Any sprites in memory will be destroyed when this command is executed.
5	Activates the MOVE CHR$ SQR CURSOR 1 place to the left - non-destructive.
6	Activates the MOVE CHR$ SQR CURSOR 1 place down - non-destructive.
7	Activates the MOVE CHR$ SQR CURSOR 1 place up - non-destructive.
8	Activates the MOVE CHR$ SQR CURSOR 1 place to the right - non-destructive.
9	Activates the SET CHR$ at current position.
0	Activates the CLEAR CHR$ SQR at current position.
% (SYMBOL SHIFT 5)	Activates the MOVE SPRITE SCREEN CURSOR 1 place to the left.
& (SYMBOL SHIFT 6)	Activates the MOVE SPRITE SCREEN CURSOR 1 place down.
' (SYMBOL SHIFT 7)	Activates the MOVE SPRITE SCREEN CURSOR 1 place up.
((SYMBOL SHIFT 8)	Activates the MOVE SPRITE SCREEN CURSOR 1 place to the right.
< (SYMBOL SHIFT R)	Activates the RELOCATE SPRITES function. Allows the user to move the sprite data about in memory, between the top of the Sprite Generator Program and address 65520. e.g. a positive number i.e. 50, moves the data 50 bytes up in memory. A negative number i.e. -50, moves the data 50 bytes down in memory. CAUTION - use this function with care.
BREAK and SPACE	Activates the PLACE SPRITE INTO SPRITE WINDOW function. This allows you to place a sprite of smaller dimensions into a sprite of greater dimensions, at a position of ROW, COL in the greater sprite in memory - the smaller sprite is left unaltered.

NOTE: ATTR = 0, Attributes of smaller sprite ignored.
ATTR = 1, Attributes of smaller sprite taken and placed with sprite.
Three options given:

1 GETBLS
GETs the smaller sprite directly into the window of the larger sprite.

2 GETORS
GETs the smaller sprite and ORs it into the window of the larger sprite.

3 GETXRS
GETs the smaller sprite and XORs it into the window of the larger sprite.

4 GETNDS
GETs the smaller sprite and ANDs it into the window of the larger sprite.

THE WHITE LIGHTNING ARCADE GRAPHICS LIBRARY

ARCADE CHARACTER NUMBER	
1- 8	Asteroids Space Ships
9- 11	Asteroids
12	Asteroids Flying Saucer
13- 20	Pac-Men
21- 22	Pac-Men Ghosts
23- 25	Fruit
26- 33	Pac-Men Maze Parts
34- 44	Assault Course type games
45- 54	Defender type games
55- 62	Defender type landscapes
63- 67	Space Invaders
68- 70	Space Invaders Bases
71- 74	Space Invaders Guns etc.
75- 84	City Bomber type games
85- 88	Lunar Lander type games
89- 98	Frogger type games
99-107	Centipede type games
108-117	War type games
118-130	Donkey Kong type games
131-136	Space War type games
137-141	Explosions
142-148	Bug-Eyed Monsters
149-152	Robots
153-158	Adventure type games Treasure
159-167	Zaps

WHITE LIGHTNING DEMONSTRATION SPRITE LIBRARY

SPRITE NUMBER	DESCRIPTION	INK COL	PAPER COL	LENGTH	HEIGHT
1	VINTAGE CAR	4	0	4	2
2	VAN	5	0	4	2
3	DRAGSTER	6	0	4	2
4	DUCK	6	0	3	3
5	DANCER	7	0	2	4
6	ROCKET	5	0	4	2
7	SPIDER #1	5	0	4	5
8	SPIDER #2	5	0	4	5
9	TOP OF TRAIN	4	0	11	2
10	RAILWAY TRACK	6	0	8	1
11	SMALL WALL	1,5	7	4	1
12	OASIS LOGO	5,7	0	12	4
13	T.V.	2	0	15	12
14	TOP OF RAILWAY COACH	5	0	10	2
15	SPACE SHIP	0	5	4	2
16	SHADOW OF SPACE SHIP	5	7	4	1

17	LARGE WALL	2,7	7	8	2
18	TRAIN WHEELS #1	4,7	0	11	1
19	TRAIN WHEELS #2	4,7	0	11	1
20	TRAIN WHEELS #3	4,7	0	11	1
21	TRAIN WHEELS #4	4,7	0	11	1
22	RAILWAY COACH #1	7	0	10	1
23	RAILWAY COACH #2	7	0	10	1
24	INVADER 0 DEGREES	6	0	2	2
25	INVADER 90 DEGREES	6	0	2	2
26	INVADER 180 DEGREES	6	0	2	2
27	INVADER 270 DEGREES	6	0	2	2
28	FACE WITH HAT #1	4	0	4	3
29	FACE WITH HAT #2	4	0	4	3
30	'WHITE'	5,7	0	7	2
31	'LIGHTNING'	5,7	0	10	2
32	LIGHTNING BOLT	5	0	13	4
33	CRAB	4,7	0	5	3
34	LUNAR LANDER	7	0	6	4
35	RADAR #1	7	0	2	1
36	RADAR #2	7	0	2	1
37	RADAR #3	7	0	2	1
38	RADAR #4	7	0	2	1
39	RADAR #5	7	0	2	1
40	RADAR #6	7	0	2	1
41	RADAR #7	7	0	2	1
42	RADAR #8	7	0	2	1
43	EXPLOSION	7	0	2	2
44	LUNAR SURFACE	6	0	15	1
45	ROTATING BALL #1	6	0	4	4
46	ROTATING BALL #2	6	0	4	4
47	ROTATING BALL #3	6	0	4	4
48	ROTATING BALL #4	6	0	4	4
49	CLOCKWORK TOYS #1	4	0	3	12
50	CLOCKWORK TOYS #2	4	0	3	12
59	'TRY THIS'	4	0	8	1
60	'WITHOUT'	4	0	8	1
61	'FROM'	5	0	5	1
251	BOUNCING BALL #1	0	6	3	3
252	BOUNCING BALL #2	0	6	3	3
253	BOUNCING BALL #3	0	6	3	3
254	BOUNCING BALL #4	0	6	3	3

This glossary contains all of the word definitions in Release 1 of fig-FORTH. The definitions are presented in the order of their ASCII sort and are reproduced courtesy of the FORTH INTEREST GROUP, P.O. BOX 1105, SAN CARLOS, CA 94070.

The first line of each entry shows a symbolic description of each action of the procedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. hi 8 bits zero)
c	7 bit ASCII character (hi 9 bits zero)
d	32 bit signed double integer, most significant portion with sign on top of stack.
f	boolean flag. 0 = false, non-zero = true.
ff	boolean false flag = 0
n	16 bit signed integer number
u	16 bit unsigned integer
tf	boolean true flag = non-zero

The capital letters on the right show definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if other than one.
E	Intended for execution only.
L0	Level zero definition of FORTH-78
L1	Level 1 definition of FORTH-78
P	Has precedence bit set. Will execute even when compiling.
U	A user variable.

Unless otherwise noted, all references to numbers are for 16 bit signed integers. The high byte of a number is on top of the stack, with the sign on the leftmost bit. For 32 bit signed double numbers, the most significant bit (with the sign) is on top.

All arithmetic is implicitly 16 bit signed integer math, with error and underflow indication specified.

NOTE: All references to disc in this documentation can be read as references to the disc simulation area in memory from C200H to F000H, which are treated as a very limited disk capacity by White Lightning, and do not in any way change the operation or description of any of the FORTH words defined in this documentation. DO NOT use DR0, DR1 or GO.

! n addr --- L0

Store 16 bits of n at address. Pronounced "store".

!CSP

Save the stack position in CSP. Used as part of the compiler security.

```
#                   d1 --- d2                     L0
```

Generate from a double number d1, the next ASCII character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S.

```
#>                   d --- addr count             L0
```

Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

```
#BUF                   --- n
```

A constant returning the number of disc buffers allocated.

```
#S                  d1 --- d2                     L0
```

Generates ascii text in the text output buffer, by the use of #, until a zero double number results. Used between <# and #>.

```
'                      --- addr                 P,L0
```

Used in the form: ' nnnn

Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in colon definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

```
(                                               P,L0
```

Used in the form: (cccc)

Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

```
(.")                                              C+
```

The run-time procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

```
(;CODE)                                            C
```

The run-time procedure, compiled by ;CODE, that re-writes the code field of the most recently defined word to point to the following machine code sequence. See ;CODE.

```
(+LOOP)              n ---                        C2
```

The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(ABORT)

Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative procedure. See WARNING.

(DO) C

The run-time procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)

Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) n1 n2 --- addr count

Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP) C2

The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr1 --- d2 addr2

Convert the ASCII text beginning at addr1 + 1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. Addr2 is the address of the first unconvertible digit. Used by NUMBER.

* n1 n2 --- prod L0

Leave the signed product of two signed numbers.

*/ n1 n2 n3 --- n4 L0

Leave the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence n1 n2 * n3 /.

*/MOD n1 n2 n3 --- n4 n5 L0

Leave the quotient n5 and remainder n4 of the operation n1*n2/n3. A 31 bit intermediate product is used as for */.

+ n1 n2 --- sum L0

Leave the sum of n1+n2.

```
+!              n addr ---                              L0
```

Add n to the value at the address. Pronounced "plus-store".

```
+-               n1 n2 --- n3
```

Apply the sign of n2 to n1, which is left as n3.

```
+BUF              add1 --- addr2 f
```

Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV.

```
+LOOP               n1 --- (run)
               addr n2 --- (compile)                P,C2,L0
```

Used in a colon-definition in the form:

```
                DO  ...  n1  +LOOP
```

At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and the execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

```
+ORIGIN              n --- addr
```

Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

```
,                    n --- ,                            L0
```

Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

```
-                n1 n2 --- diff                         L0
```

Leave the difference of n1-n2.

```
-->                                                   P,L0
```

Continue interpretation with the next screen. (Pronounced next-screen).

```
-DUP                n1 --- n1     (if zero)
                    n1 --- n1 n1  (non-zero)            L0
```

Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

```
-FIND                  --- pfa b tf  (found)
                       --- ff        (not found)
```

Accepts the next text word (delimited by blanks) in the input stream to HERE, then searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

```
-TRAILING      addr n1 --- addr n2
```

Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n1 to addr+n2 are blanks.

```
.                    n ---                               L0
```

Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blanks follows. Pronounced "dot".

```
."                                                     P,L0
```

Used in the form: ." cccc "

Compiles an in-line string cccc (delimited by the trailing "), with an execution procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". See (.").

```
.LINE         line scr ---
```

Print on the terminal device, a line of text by its line and screen number. Trailing blanks are suppressed.

```
.R               n1 n2 ---
```

Print the number n1 right aligned in a field whose width is n2. No following blanks printed.

```
/                n1 n2 --- quot                          L0
```

Leave the signed quotient of n1/n2.

```
/MOD             n1 n2 --- rem quot                      L0
```

Leave the remainder and signed quotient of n1/n2. The remainder has the sign of the dividend.

```
0 1 2 3                --- n
```

These small numbers are used so often, that it is attractive to define them by name in the dictionary as constants.

0<   n --- f   L0

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0=   n --- f   L0

Leave a true flag is the number is equal to zero, otherwise leave a false flag.

0BRANCH   f ---   C2

The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+   n1 --- n2   L1

Increment n1 by 1.

2+   n1 --- n2   L1

Leave n1 incremented by 2.

2!   nlow nhigh addr ---

32 bit store, nhigh is stored at addr; nlow is stored at addr+2.

2@   addr --- nlow nhigh

32 bit fetch, nhigh is fetched from addr; nlow is fetched from addr-2.

2DUP   n2 n1 --- n2 n1 n2 n1

Duplicates the top two values on the stack. Equivalent to OVER OVER.

:   P,E,L0

Used in the form called a colon-definition:

```
               : cccc  ...  ;
```

Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '...' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

;   P,C,L0

Terminate a colon-definition and stop further compilation. Compiles the run-time ;S.

```
;CODE                                                      P,C,L0

Used in the form:
               : cccc  ....  ;CODE
               assembly mnemonics
Stop compilation and terminate a new defining word cccc by compiling (;CODE).  Set
the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following
mnemonics.  This facility is included for those users who may wish to write a Z80
Assembler in FORTH.

When cccc later executes in the form:
                  cccc   nnnn
the word nnnn will be created with its execution procedure given by the machine
code following cccc.  That is, when nnnn is executed, it does so by jumping to the
code after nnnn.  An existing defining word must exist in cccc prior to ;CODE.


;S                                                           P,L0

Stop interpretation of a screen.  ;S is also the run-time word compiled at the end
of a colon-definition, which returns execution to the calling procedure.


<                n1 n2 --- f                                   L0

Leave a true flag if n1 is less than n2; otherwise leave a false flag.


<#                                                             L0

Setup for pictured numeric output formatting using the words:
               <#  #  #S  SIGN  #>
The conversion is done on a double number producing text at PAD.


<BUILDS                                                      C,L0

Used within a colon-definition:
             :  cccc  <BUILDS  ...
                      DOES>    ...  ;
Each time cccc is executed, <BUILDS defines a new word with a high-level execution
procedure.  Executing cccc in the form:
                cccc  nnnn
uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part
for nnnn.  When nnnn is later executed, it has the address of its parameter area
on the stack and executes the words after DOES> in cccc.  <BUILDS and DOES> allow
run-time procedures to written in high-level, rather than in assembler code (as
required by ;CODE).


=                n1 n2 --- f                                   L0

Leave a true flag if n1=n2 otherwise leave a false flag.


>                n1 n2 --- f                                   L0

Leave a true flag if n1 is greater than n2 otherwise a false flag.
```

>R n --- C,L0

Remove a number from the computation stack and place as the most accessible on the return stack. Use should be balanced with R> in the same definition.

? addr --- L0

Print the value contained at the address in free format according to the current base.

?COMP

Issue error message if not compiling.

?CSP

Issue error message if stack position differs from value saved in CSP.

ERROR f n ---

Issue an error message number n, if the boolean flag is true.

?EXEC

Issue an error message if not executing.

?LOADING

Issue an error message if not loading

?PAIRS n1 n2 ---

Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

Issue an error message is the stack is out of bounds.

?TERMINAL --- f

Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation.

@ addr --- n L0

Leave the 16 bit contents of address.

```
ABORT                                                    L0
```

Clear the stacks and enter the execution state. Return control to the operator's terminal, printing a message appropriate to the installation.

```
ABS                     n --- u                          L0
```

Leave the absolute value of n as u.

```
AGAIN           addr n --- (compiling)              P,C2,L0
```

Used in a colon-definition in the form:

```
                BEGIN  ...  AGAIN
```

At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> is executed one level below).

At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

```
ALLOT                   n ---                            L0
```

Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory. n is with regard to computer address type (byte or word).

```
AND              n1 n2 --- n2                            L0
```

Leave the bitwise logical "AND" of n1 and n2 as n3.

```
B/BUF                     --- n
```

This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

```
B/SCR                     --- n
```

This component leaves the number of blocks per editing screen. By convention, an editing screen is 512 bytes, organised as 8 lines of 64 characters each.

```
BACK              addr ---
```

Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

```
BASE                      --- addr
```

A user variable containing the current number base used for input and output conversion.

```
BEGIN                   --- addr n (compilation)        P,L0
```

Occurs in a colon-definition in form:

```
              BEGIN  ...  UNTIL
              BEGIN  ...  AGAIN
              BEGIN  ...  WHILE  ...  REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT, a return to BEGIN always occurs.

At compile time BEGIN leaves its return address and n for compiler error checking.

```
BL                      --- c
```

A constant that leaves the ASCII value for blank.

```
BLANKS      addr count ---
```

Fill in an area of memory beginning at addr with blanks.

```
BLK                     --- addr                          L0
```

A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

```
BLOCK                 n --- addr                          L0
```

Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to whichever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is re-written to disc before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH.

```
BRANCH                                                  C2,L0
```

The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.

```
BUFFER                n --- addr
```

Obtain the next memory buffer, assigning it to block n. If the contents of the buffer are marked up as updated, it is written to the disc. The block is not read from the disc. The address left is the first cell within the buffer for data storage.

```
C!              b addr ---
```

Store 8 bits at address.

```
C,                    b ---
```

Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.

```
C@                 addr --- b
```

Leave the 8 bit contents of memory address.

```
CASE                    --- n (compiling)
```

Occurs in a colon definition in the form:

```
            CASE
            n OP ..... ENDOF
            .....
            ENDCASE
```

At run-time, CASE marks the start of a sequence of OF ... ENDOF statements.

At compile time CASE leaves n for compiler error checking.

```
CFA                 pfa --- cfa
```

Convert the parameter field address of a definition to its code field address.

```
CMOVE    from to count ---
```

Move the specified quantity of bytes beginning at address 'from' to address 'to'. The contents of address 'from' are moved first proceeding toward high memory.

```
COLD
```

The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

```
COMPILE                                                    C2
```

When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

```
CONSTANT              n ---                                L0
```

A defining word used in the form:

```
              n  CONSTANT  cccc
```

to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n to the stack.

```
CONTEXT                 --- addr                         U,L0
```

A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 --- addr2 n L0

Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and that the actual text starts with the second byte. Typically, COUNT is followed by TYPE.

CR L0

Transmit a carriage return and line feed to the selected output device.

CREATE

A defining word used in the form:

```
               CREATE cccc
```

by such words as CODE and CONSTANT to create a dictionary header for a Forth definition. The code field contains the address of the word's parameter field. A new word is created in the CURRENT vocabulary.

CSP --- addr U

A user variable temporarily storing the stack pointer position, for compilation error checking.

D+ d1 d2 --- dsum

Leave the double number sum of two double numbers.

D+- d1 n --- d2

Apply the sign of n to the double number d1, leaving it as d2.

D. d --- L1

Print a signed double number from a 32 bit two's complement value. The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current base. A blank follows. Pronounced D-dot.

D.R d n --- L0

Print a signed double number d right aligned in a field n characters wide.

DABS d --- ud

Leave the absolute value of a double number.

DECIMAL L0

Set the numeric conversion BASE for decimal input-output.

```
DEFINITIONS                                            L1
```

Used in the form:

```
              cccc DEFINITIONS
```

Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it in the CONTEXT vocabulary, and executing DEFINITIONS made both specify vocabulary cccc.

```
DIGIT             c n1 --- n2 tf  (ok)
                  c n1 --- ff     (bad)
```

Converts the ASCII characters c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

```
DLITERAL             d --- d  (executing)
                     d ---    (compiling)                 P
```

If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

```
DMINUS              d1 --- d2
```

Convert d1 to its double number two's complement.

```
DO                n1 n2 --- (execute)
                addr n --- (compile)                P,C2,L0
```

Occurs in a colon-definition in form:

```
              DO  ...  LOOP
              DO  ... +LOOP
```

At run time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop, 'I' will copy the current value of the index to the stack. See I, LOOP, +LOOP, LEAVE.

When compiling within the colon definition, DO compiles (DO), leaving the following address addr and n for later error checking.

```
DOES>                                                  L0
```

A word which defines the run-time action within a high-level defining word. DOES> alters the code field and first parameter of the new word, to execute the sequence of compiled word addresses following DOES>. Used in combination with <BUILDS. When the word DOES> part executes, it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, multi-dimensional arrays and compiler generation.

```
DP                      --- addr                          U,L
```

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

```
DPL                     --- addr                         U,L0
```

A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used to hold output column location of a decimal point, in user generated formatting. The default value on single number input is -1.

```
DROP                  n ---                                L0
```

Drop the number from the stack.

```
DUMP               addr ---                                L0
```

Print the contents of n memory locations beginning at addr. Both addresses and contents are shown in the current numeric base.

```
DUP                   n --- n n                            L0
```

Duplicate the value on the stack.

```
ELSE          addr1 n1 --- addr2 n2
                    (compiling)                      P,C2,L0
```

Occurs within a colon-definition within the form:

```
              IF  ...  ELSE  ...  ENDIF
```

At run-time, ELSE executes after the true part following IF. ELSE forces the execution to skip over the following false part, and resumes execution after the ENDIF. It has no stack effect.

A compile time ELSE emplaces branch reserving a branch offset, leaves the address addr2 and n2 for error treating. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

```
EMIT                  c ---                                L0
```

Transmit ASCII character c to the selected output device. OUT is incremented for each character output.

```
EMPTY-BUFFERS                                              L0
```

Mark all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disc. This is also an initialization procedure before first use of the disc.

ENCLOSE   addr1 c --- addr1 n1 n2 n3

The text scanning primitive used by WORD. From the text address addr1 and an ASCII delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ASCII 'null', treating it as an unconditional delimiter.

END   P,C2,L0

This is an 'alias' or duplicate definition for UNTIL.

ENDCASE   addr n --- (compile)

Occurs in a colon definition in the form:

```
          CASE
          n OF ..... ENDOF
          .....
          ENDCASE
```

At run-time ENDCASE marks the conclusion of a CASE statement.

At compile time ENDCASE computes forward branch offsets.

ENDIF   addr n --- (compile)   P,C0,L0

At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in Fig-FORTH. See also IF and ELSE.

At compile time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ENDOF   addr n --- (compile)

Used as ENDIF but in CASE statements.

ERASE   addr n ---

Clear a region of memory to zero from addr over n addresses.

ERROR   line --- in blk

Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 4 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 4. If WARNING=0, n is just printed as a message number (non disc installation). If warning is -1, the definition ABORT is executed, which executes the system ABORT. The user may cautiously modify this by altering (ABORT). Fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

```
EXECUTE           addr ---
```

Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

```
EXPECT      addr count ---                               L0
```

Transfer characters from the terminal to address, until a return or the count of characters has been received. One or more nulls are added at the end of the text.

```
FENCE                  --- addr                           U
```

A user variable containing an address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE.

```
FILL       addr quan b ---
```

Fill memory at the address with the specified quantity of bytes b.

```
FIRST                  --- n
```

A constant that leaves the address of the first (lowest) block buffer.

```
FLD                    --- addr                           U
```

A user variable for control of number output field width. Presently unused in Fig-FORTH.

```
FORGET                                                  E,L0
```

Deletes definition named cccc from the dictionary with all entries physically following it. In Fig-FORTH, an error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

```
FORTH                                                   P,L1
```

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile time.

```
HERE                   --- addr                          L0
```

Leave the address of the next available dictionary location.

```
HEX                                                      L0
```

Set the numeric conversion base to sixteen (hexadecimal).

```
HLD                     --- addr                           L0
```

A user variable that holds the address of the latest character of text during numeric output conversion.

```
HOLD                  c ---                                L0
```

Used between <# and #> to insert an ASCII character into a pictured numeric output string.

e.g. 2E HOLD will place a decimal point.

```
I                       --- n                            C,L0
```

Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependent. See R.

```
ID.              addr ---
```

Print a definition's name from its name field address.

```
IF                    f ---         (run-time)
                        --- addr n  (compile)       P,C2,L0
```

Occurs in a colon-definition in the form:

```
              IF  (tp)  ...  ENDIF
              IF  (tp)  ...  ELSE  (fp)  ...  ENDIF
```

At run-time, IF selects execution based on a boolean flag. If f is a true (non-zero), execution continues ahead through the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional; if missing, false execution skips to just after ENDIF.

At compile time, IF compiles 0BRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

```
IMMEDIATE
```

Mark the most recently made definition so that when encountered at compile time it will be executed rather than compiled, i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of an immediate definition by preceding it with [COMPILE].

```
IN                      --- addr                           L0
```

A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX from to ---

Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disc screens.

INTERPRET

The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT, it is converted to a number according to the current base. That also failing, an error message echoing the name with a "?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER.

KEY --- c L0

Leave the ASCII value of the next terminal key struck.

LATEST --- addr

Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE C,L0

Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa --- lfa

Convert the parameter field address of a dictionary definition to its link field address.

LIMIT --- n

A constant leaving the address just above the highest memory available for a disc buffer. Usually this is the highest system memory.

LINE n --- addr

Leave address of line n of current screen. This address will be in the disc buffer area

LIST n --- L0

Display the ASCII text of screen n on the selected output device. SCR contains the screen number during and after this process.

```
LIT                     --- n                          C,L0
```

Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

```
LITERAL               n --- (compiling)              P,C2,L0
```

If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
              :  xxx   (calculate) LITERAL  ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and LITERAL compiles this value.

```
LOAD                  n ---                             L0
```

Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->.

```
LOOP             addr n --- (compiling)              P,C2,L0
```

Occurs in a colon-definition in form:

```
              DO  ...  LOOP
```

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile-time. LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

```
M*                n1 n2 --- d
```

A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

```
M/                 d n1 --- n2 n3
```

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

```
M/MOD            ud1 u2 --- u3 ud4
```

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

```
MAX               n1 n2 --- max                         L0
```

Leaves the greater of two numbers.

MESSAGE   n ---

Print on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc unavailable).

MIN   n1 n2 --- min   L0

Leave the smaller of two numbers.

MINUS   n1 --- n2   L0

Leave the two's complement of a number.

MOD   n1 n2 --- mod   L0

Leave the remainder of n1/n2, with the same sign as n1.

NEXT

This is the inner interpreter that uses the interpretive IP to execute compiled Forth definitions. It is not directly executed but is the return point for all code procedures. It acts by fetching the address pointed by IP, and storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of Forth.

NFA   pfa --- nfa

Convert the parameter field address of a definition to its name field. See PFA.

NUMBER   addr --- d

Convert a character string left at addr with a preceding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OFFSET   --- addr   U

A user variable which may contain a block offset to disc drives. The contents of OFFSET is added to the stack number by BLOCK. Messages by MESSAGE are independent of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

OR   n1 n2 --- or   L0

Leave the bit-wise logical "OR" of two 16 bit values.

```
OUT                     --- addr                           U
```

A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.

```
OVER              n1 n2 --- n1 n2 n1                      L0
```

Copy the second stack value, placing it as the new top.

```
PAD                     --- addr                          L0
```

Leave the address of the text output buffer, which is a fixed offset above HERE.

```
PFA                 nfa --- pfa
```

Convert the name field address of a compiled definition to its parameter field address.

POP

The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is s Forth re-entry point after machine code.

```
PREV                    --- addr
```

A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.

PUSH

This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

PUT

This code sequence stores machine register contents over the topmost computation value and returns to NEXT. It is not directly executable, but is a Forth re-entry point after machine code.

QUERY

Input 80 characters of text (or until a "return") from the operator's terminal. Text is positioned at the address contained in TIB with IN set to zero.

```
QUIT                                                      L1
```

Clear the return stack, stop compilation, and return control to the operator's terminal. No message is given.

```
R                      --- n
```

Copy the top of the return stack to the computation stack.

```
R#                     --- addr                           U
```

A user variable which may contain the location of an editing cursor, or other file related function.

```
R/W         addr blk f ---
```

The Fig-FORTH standard read-write linkage. addr specifies the source or destination block buffer. blk is the sequential number of the referenced block; and f is a flag for f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and any error checking.

```
R>                     --- n                             L0
```

Remove the top value from the return stack and leave it on the computation stack. See >R and R.

```
R0                     --- addr                           U
```

A user variable containing the initial location of the return stack. Pronounced R-zero. See RP!

```
REPEAT          addr n --- (compiling)                 P,C2
```

Used within a colon-definition in the form:

```
                BEGIN  ...  WHILE  ...  REPEAT
```

At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

```
ROT           n1 n2 n3 --- n2 n3 n1                      L0
```

Rotate the top three values on the stack, bringing the third to the top.

```
RP@                 addr
```

Leaves the current value in the return stack pointer register.

```
RP!
```

A computer dependent procedure to initialise the return stack pointer from user variable R0.

```
S->D                 n --- d
```

Sign extend a single number to form a double number.

S0 --- addr U

A user variable that contains the initial value for the stack pointer. Pronounced S-zero. See SP!

SCR --- addr U

A user variable containing the screen number most recently reference by LIST.

SIGN n d --- d L0

Stores an ASCII "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #>.

SMUDGE

Used during word definition to toggle the "smudge bit" in a definition's name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!

A computer dependent procedure to initialize the stack pointer from S0.

SP@ --- addr

A computer dependent procedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed. (e.g. 1 2 SP@ @ . . . would print 2 2 1).

SPACE

Transmit an ASCII blank to the output device.

SPACES n --- L0

Transmit n ASCII blanks to the output device.

STATE --- addr L0,U

A user variable containing the compilation state. A non-zero indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 --- n2 n1 L0

Exchange the top two values On the stack.

TASK

A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

TEXT   c ---

Accept the following test to PAD. c is the text delimiter.

THEN   P,C0,L0

An alias for ENDIF.

TIB   --- addr   U

A user variable containing the address of the terminal input buffer.

TOGGLE   addr b ---

Complement the contents of addr by the bit pattern b.

TRAVERSE   addr1 n --- addr2

Move across the name field of a Fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=-1, the motion is toward low memory. The addr2 resulting is the address of the other end of the name.

TYPE   addr count ---   L0

Transmit count characters from addr to the selected output device.

U<   u1 u2 --- f

Leave the boolean value of an unsigned less-than comparison. Leaves f=1 for u1 > u2; otherwise leaves 0. This function should be used when comparing memory addresses.

U*   u1 u2 --- ud

Leave the unsigned double number product of two unsigned numbers.

U.   u ---

Prints an unsigned 16 bit number converted according to BASE. A trailing blank follows.

```
U/                ud u1 --- u2 u3
```

Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

```
UNTIL                 f --- (run-time)
                addr n --- (compile)                P,C2,L0
```

Occurs within a colon-definition in the form:

```
                         BEGIN ... UNTIL
```

At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN, if true, execution continues ahead.

At compile-time, UNTIL compiles (0BRANCH) and an offset from HERE to addr. n is used for error tests.

```
UPDATE                                                   L0
```

Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred to disc should its buffer be required for storage of a different block.

```
USE                     --- addr
```

A variable containing the address of the block buffer to use next, as the least recently written.

```
USER                  n ---                              L0
```

A defining word used in the form:

```
                   n USER cccc
```

which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this upper variable. When cccc is later executed, it places the sum of its offset and the user base address on the stack, as the storage address of that particular variable.

```
VARIABLE                                               E,L0
```

A defining word used in the form:

```
                   n VARIABLE cccc
```

When VARIABLE is executed, it creates the definition cccc with its parameter field initialised to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

```
VOC-LINK                --- addr                          U
```

A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields, to allow control for FORGETting through multiple vocabularies.

```
VOCABULARY                                                  E,L
```

A defining word used in the form:

```
               VOCABULARY cccc
```

to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary, into which, new definitions are placed.

In Fig-FORTH, cccc will also be chained so as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK.

VLIST

List the names of the definitions in the context vocabulary. Pressing "Break" will terminate the listing.

```
WARNING                --- addr                            U
```

A user variable containing a value controlling messages.
If = 1 disc is present, and screen 4 of drive 0 is the base location for messages.
If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified procedure. See MESSAGE, ERROR, ABORT.

```
WHERE            n1 n2 ---
```

If an error occurs during LOAD from disc, ERROR leaves these values on the stack to show the user where the error occurred. WHERE uses these to print the screen and line number of where this is.

```
WHILE                    f --- (run-time)
              addr1 n1 --- addr1 n1 addr2 n2            P,C2
```

Occurs in a colon-definition in the form:

```
                  BEGIN ... WHILE (tp) ... REPEAT
```

At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part through to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE emplaces (0BRANCH) and leaves addr2 of the reserved offset. The stack values will be resolved by REPEAT.

```
WIDTH                  --- addr                            U
```

In Fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definitions name. It must be 1 through to 31, having a default value of 31. The name character count and its natural characters are saved, up to the value of WIDTH. The value may be changed at any time within the above limits.

```
WORD                c ---                               L0
```

Read the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurrences of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X

This is pseudonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the execution procedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

```
XOR              n1 n2 --- xor                          L1
```

Leave the bit-wise logical Exclusive-OR of two values.

```
[                                                     P,L1
```

Used in a colon-definition in the form:

```
                  : xxx   words   more   ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

```
[COMPILE]                                              P,C
```

Used in a colon-definition in the form:

```
                  : xxx [COMPILE] FORTH  ;
```

[COMPILE] will force the compilation of an immediate definition, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile time.

```
]                                                       L1
```

Resume compilation, to the completion of a colon-definition. See [.

ADDITIONAL GLOSSARY

```
C/L                     --- n
```

A constant containing the number of characters per line (64).

```
WARM                    ---
```

This will perform a warm-start.

NOOP ---

This will perform a no-operation, i.e. do nothing.

WARM->COLD

This allows you to preserve any FORTH word defined to date, so that a COLD start will not delete them. When saving your code, save from 24832 to HERE.

e.g. `: NEWWORD ." THIS WILL BE PRESERVED BY WARM->COLD " ;`

If we now do a COLD start this will be lost, but if we first key in WARM->COLD and then do a COLD start, it will still be there.

WORD	PARAMETERS	ACTION
WCRV	HGT, LEN, COL, ROW, NPX	Scroll the window vertically with wrap by NPX pixels.
SCRV	HGT, LEN, COL, ROW, NPX	Scroll the window vertically without wrap by NPX pixels.
WRR1V	HGT, LEN, COL, ROW	Scroll the window 1 pixel right with wrap.
WRL1V	HGT, LEN, COL, ROW	Scroll the window 1 pixel left with wrap.
WRR4V	HGT, LEN, COL, ROW	Scroll the window 4 pixels right with wrap.
WRL4V	HGT, LEN, COL, ROW	Scroll the window 4 pixels left with wrap.
WRR8V	HGT, LEN, COL, ROW	Scroll the window 8 pixels right with wrap.
WRL8V	HGT, LEN, COL, ROW	Scroll the window 8 pixels left with wrap.
SCR1V	HGT, LEN, COL, ROW	Scroll the window 1 pixel right without wrap.
SCL1V	HGT, LEN, COL, ROW	Scroll the window 1 pixel left without wrap.
SCR4V	HGT, LEN, COL, ROW	Scroll the window 4 pixels right without wrap.
SCL4V	HGT, LEN, COL, ROW	Scroll the window 4 pixels left without wrap.
SCR8V	HGT, LEN, COL, ROW	Scroll the window 8 pixels right without wrap.
SCL8V	HGT, LEN, COL, ROW	Scroll the window 8 pixels left without wrap.
ATTRV	HGT, LEN, COL, ROW	Scroll the window attributes 1 character right with wrap.
ATTLV	HGT, LEN, COL, ROW	Scroll the window attributes 1 character left with wrap.
ATTUPV	HGT, LEN, COL, ROW	Scroll the window attributes 1 character up with wrap.
ATTDNV	HGT, LEN, COL, ROW	Scroll the window attributes 1 character down with wrap.
WCRM	SPN	Scroll the Sprite vertically with wrap by NPX pixels.

SCRM	SPN	Scroll the Sprite vertically without wrap by NPX pixels.
WRR1M	SPN	Scroll the Sprite 1 pixel right with wrap.
WRL1M	SPN	Scroll the Sprite 1 pixel left with wrap.
WRR4M	SPN	Scroll the Sprite 4 pixels right with wrap.
WRL4M	SPN	Scroll the Sprite 4 pixels left with wrap.
WRR8M	SPN	Scroll the Sprite 8 pixels right with wrap.
WRL8M	SPN	Scroll the Sprite 8 pixels left with wrap.
SCR1M	SPN	Scroll the Sprite 1 pixel right without wrap.
SCL1M	SPN	Scroll the Sprite 1 pixel left without wrap.
SCR4M	SPN	Scroll the Sprite 4 pixels right without wrap.
SCL4M	SPN	Scroll the Sprite 4 pixels left without wrap.
SCR8M	SPN	Scroll the Sprite 8 pixels right without wrap.
SCL8M	SPN	Scroll the Sprite 8 pixels left without wrap.
ATTRM	SPN	Scroll the Sprite attributes 1 character right with wrap.
ATTLM	SPN	Scroll the Sprite attributes 1 character left with wrap.
ATTUPM	SPN	Scroll the Sprite attributes 1 character up with wrap.
ATTDNM	SPN	Scroll the Sprite attributes 1 character down with wrap.
GETBLS	SPN, COL, ROW	Block move screen data from screen to Sprite.
GETXRS	SPN, COL, ROW	Logically XOR screen data into Sprite data.
GETORS	SPN, COL, ROW	Logically OR screen data into Sprite data.
GETNDS	SPN, COL, ROW	Logically AND screen data into Sprite data.
PUTBLS	SPN, COL, ROW	Block move Sprite data from Sprite to screen.
PUTXRS	SPN, COL, ROW	Logically XOR Sprite data into screen data.
PUTORS	SPN, COL, ROW	Logically OR Sprite data into screen data.
PUTNDS	SPN, COL, ROW	Logically AND Sprite data into screen data.
GWBLS	SPN, COL, ROW, SCOL, SROW, HGT, LEN.	Block move screen data from screen window into Sprite window.

GWXRS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically XOR screen data from screen window into Sprite window.
GWORS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically OR screen data from screen window into Sprite window.
GWNDS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically AND screen data from screen window into Sprite window.
GWATTS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Block move attributes from screen window into Sprite window.
PWBLS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Block move Sprite data from Sprite window into screen window.
PWXRS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically XOR Sprite window data into screen window.
PWORS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically OR Sprite window data into screen window.
PWNDS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Logically AND Sprite window data into screen window.
PWATTS	SPN, COL, ROW, SCOL, SROW, HGT, LEN	Block move Sprite window attributes into screen window.
GWBLM	SP1, SP2, SCOL, SROW	Block move Sprite SP1 into Sprite SP2 at SCOL,SROW.
GWXRM	SP1, SP2, SCOL, SROW	Logically XOR Sprite SP1 into Sprite SP2 at SCOL,SROW.
GWORM	SP1, SP2, SCOL, SROW	Logically OR Sprite SP1 into Sprite SP2 at SCOL,SROW.
GWNDM	SP1, SP2, SCOL, SROW	Logically AND Sprite SP1 into Sprite SP2 at SCOL,SROW.
GWATTM	SP1, SP2, SCOL, SROW	Block move attributes of Sprite SP1 into Sprite SP2 at SCOL,SROW.
PWBLM	SP1, SP2, SCOL, SROW	Block move window at SCOL,SROW of Sprite SP2 into Sprite SP1.
PWXRM	SP1, SP2, SCOL, SROW	Logically XOR window at SCOL,SROW of Sprite SP2 into Sprite SP1.
PWORM	SP1, SP2, SCOL, SROW	Logically OR window at SCOL,SROW of Sprite SP2 into Sprite SP1.

PWNDM	SP1, SP2, SCOL, SROW	Logically AND window at SCOL,SROW of Sprite SP2 into Sprite SP1.
PWATTM	SP1, SP2, SCOL, SROW	Block move attributes of window at SCOL,SROW of Sprite SP2 into Sprite SP1.
COPYM	SP1, SP2	As GWBLM but SCOL,SROW assumed zero.
COPXRM	SP1, SP2	As GWXRM but SCOL,SROW assumed zero.
COPORM	SP1, SP2	As GWORM but SCOL,SROW assumed zero.
COPNDM	SP1, SP2	As GWNDM but SCOL,SROW assumed zero.
COPATTM	SP1, SP2	As GWATTM but SCOL,SROW assumed zero.
INVV	HGT, LEN, COL, ROW	Invert screen window.
MIRV	HGT, LEN, COL, ROW	Mirror screen window about its centre.
MARV	HGT, LEN, COL, ROW	Mirror screen window attributes about centre.
INVM	SPN	Invert Sprite data.
MIRM	SPN	Mirror Sprite about its centre.
MARM	SPN	Mirror Sprite attributes about centre.
SPINM	SP1, SP2	Rotate Sprite SP2 90 degrees clockwise into Sprite SP1.
DSPM	SP1, SP2	Enlarge Sprite SP2 into Sprite SP1.
HALT		Suspend CPU operation until next interrupt.
EI		Enable interrupt.
DI		Disable interrupt.
EXX		Exchange Ideal variables with the alternate Ideal variables.
INT-ON	FORTH WORD	Execute specified Forth Word under interrupt.
INT-OFF		Terminate execution of interrupt driven word.
PROG		Enter BASIC.
RESERVE	N1	Reserve N1 bytes in the dictionary for BASIC source.
GOTO	N1	Begin execution of BASIC at line N1.
RETUSR		Return to BASIC from RANDOMIZE USR 30000 call.

DSPRITE	SPN	Delete Sprite and recover bytes from below.
ISPRITE	SPN, HGT, LEN	Create Sprite and move current Sprites down to accommodate.
WIPE	SPN	Delete Sprite and recover bytes from above.
SPRITE	SPN, HGT, LEN	Create Sprite at free space after last Sprite.
RELOCATE	MLEN	Relocate Sprite space by signed 16 bit length MLEN.
COLD#	SPST, SLEN	Reset Sprite space to begin at SPST with SLEN bytes cleared to zeros.
SETAV	HGT, LEN, COL, ROW	Fill the screen window with the current attributes.
SETAM	SPN	Fill the Sprite with the current attributes.
CLSV	HGT, LEN, COL, ROW	Clear the screen window and fill with the current attributes.
CLSM	SPN	Clear the Sprite.
ADJV	HGT, LEN, COL, ROW	Adjust the screen window to lie on the screen.
ADJM	SPN, COL, ROW	Adjust COL, ROW, HGT, LEN, SCOL, SROW such that GETS and PUTS lie on the screen.
RND	N1	Leave a random number between 0 and N1 on the stack.
OUT#	N1, N2	Output LSB of N1 to 16 bit port address N2.
IN#	N1	Leave on the stack, byte from 16 bit port address N1.
ZAPINT		Create run time program with interrupt facility.
ZAP		Create run time program without interrupt facility.
CALL	N1	Execute machine code subroutine at address N1.
KB	N1, N2	Test for key press at row N1, col N2 and stack true or false flag.
SCANV	COL, ROW	The character position is scanned for screen data and a true or false flag stacked.
SCANM	SPN	The Sprite is scanned for data and a true or false flag stacked.
BLEEP	N1, N2	Sinclair BEEP. N1 is duration, N2 is pitch.

ATTON	Enable attribute switch.
ATTOFF	Disable attribute switch.

FORTH/BASIC GLOSSARY

WORD	**PARAMETERS**	**ACTION**
COPY		Copy screen to ZX-Printer.
AT	N1, N2	Move print position to N1,N2.
BORDER	N1	Set border colour to N1.
CLS		Clear whole screen, home cursor and fill with current attributes.
DRAW-ARC	N1, N2, N3	±X,±Y,ANGLE. As Sinclair's own.
CIRCLE	N1, N2, N3	X,Y,RADIUS. As Sinclair's own.
DRAW	N1, N2	±X,±Y. As Sinclair's own.
PLOT	N1, N2	X,Y. As Sinclair's own.
SCREEN$	N1, N2	Leave on the stack the ASCII code of the character at ROW N1, COL N2.
ATTR	N1, N2	Leave on the stack the attribute code of the character at ROW N1, COL N2.
POINT	N1, N2	Test pixel at N1,N2 and leave a true or false flag on the stack.
TAB	N1	Set print position to COL N1.
OVER	N1	Zero or one, as Sinclair's own.
INVERSE	N1	Zero or one, as Sinclair's own.
BRIGHT	N1	Zero or one, as Sinclair's own.
FLASH	N1	Zero or one, as Sinclair's own.
PAPER	N1	Set paper colour, as Sinclair's own.
INK	N1	Set ink colour, as Sinclair's own.

USR CALLS

PRINT USR 24832	Enter Forth from BASIC via a COLD START.
PRINT USR 24836	Enter Forth from BASIC via a WARM START.
PRINT USR 30006	Re-enter Forth from BASIC and continue execution of the next Forth word.
RANDOMIZE USR 30000	Call Forth and continue execution up to the first occurrence of the Forth word RETUSR.

EXTENDED SPECTRA FORTH GLOSSARY

WORD	**PARAMETERS**	**ACTION**
PRT-ON		Send all subsequent output to the printer.
PRT-OFF		Send all subsequent output to the screen.
EDIT	N1	Edit line number N1 from the current screen.
WARM->COLD		Create extended Forth.
J		Copy second loop index to the top of the stack.
K		Copy third loop index to the top of the stack.
DUMP	N1	Memory dump from address N1.
WARM		Perform a warm start.
EMITC	N1	As EMIT but control characters are also supported.

THE DEMONSTRATION PROGRAM - A BRIEF DESCRIPTION

THE WHITE LIGHTNING SCREEN

The green text at the top of the screen, sprites 59 and 60, is scrolled right in background.

The lightning bolt, sprite 32, is put to the screen using PUTBLS. It is then mirrored and then mirrored again using MIRV. It is removed from the screen using PUTXRS.

Two windows are defined over the 'WHITE LIGHTNING', sprites 30 and 31. The left window is scrolled left and the right window scrolled right using SCL8V and SCR8V respectively.

THE TRAIN

The steam engine is comprised of sprites 9, with sprites 18, 19, 20, and 21 for the wheels in their four positions. The coaches are comprised of sprites 14 with sprites 23 and 24 for the wheels.

The track, sprite 10, is scrolled left using WRL1V. By means of an increasing and decreasing delay loop, acceleration and deceleration effects are achieved.

THE SPIDERS

Five spiders, sprite 7, are placed on the screen. From the left, spider 1 is scrolled down by 1 pixel, spider 3 is scrolled up 8 pixels and spider 5 is scrolled up 4 pixels all - in background.

Spiders 2 and 4 are animated up and down using sprites 7 and 8.

THE SIDEWAYS SCROLLING CIRCLE OF INVADERS

Twelve Invaders, sprite 24, are placed in a circle on the screen, using either WRR1M, WRR4M or WRR8M they are scrolled in memory and then placed on the screen using PUTBLS.

The Invader in the centre of the screen is scrolled left by 1 pixel in background.

THE ARRAY OF CLOCKWORK TOYS

The green clockwork toys are animated using sprites 49 and 50. Each sprite in the array is individually placed on the screen using PUTBLS. The movement is controlled by simple 'DO LOOPS'.

THE VERTICAL ATTRIBUTE SCROLL

Using ATTUPV and decreasing and increasing delay loops the attributes placed on the screen are scrolled up, whilst a random border colour change is executed.

THE THREE VEHICLE SCROLLING DEMO

This demonstrates the 3 precisions of scrolls available - 1, 4 and 8 pixels.

Sprite 1, the vintage car, demonstrates the fine 1 pixel scroll using SCL1V.

Sprite 2, the van, demonstrates the faster 4 pixel scroll using SCL4V.

Sprite 3, the dragster, demonstrates, the very fast 8 pixel, or 1 character scroll, using SCL8V.

THE TELEVISION

The television, sprite 13, is placed on the screen, a window is defined inside the screen.

Sprite 5, the dancer is used to demonstrate the 1 pixel scroll with wrap, WRR1V.

Sprite 4, the duck, is used to demonstrate the 4 pixel scroll with wrap, WRR4V.

Sprite 6, the rocket, is used to demonstrate the 8 pixel scroll with wrap, WRR8V.

THE THREE SPACESHIPS

In this demonstration the 3 spaceships, sprite 15, are placed on the screen along with their shadows, sprite 16.

They are scrolled to the right by 1 pixel, with alternate 1 pixel up and down scrolls, to give a sense of motion.

The attributes of the foreground, sprite 17, and the background, sprite 11, are scrolled to the left. The background being scrolled at one character per execution with the foreground being scrolled two characters per execution to give a sense of perspective.

THE BOUNCING MAN WITH HAT

The bouncing man is animated using sprites 28 and 29. The sprites are placed on the screen using PUTXRS, and removed again using PUTXRS, such that the character appears to move behind the 'WHITE LIGHTNING' text without destroying it.

Simple DO LOOPS control his path.

Disabling interrupts, using DI, during the animation appeared to reduce the slight flicker of the character, the interrupts where enabled again using EI.

THE CIRCLE OF ROTATING INVADERS

From the original invader, sprite 24, sprites 25, 26 and 27 were created using the 'ROTATE' facility of the sprite development package.

This demo shows how the centre invader, scrolling left by 1 pixel in background, moves at a constant smooth rate, independent from that of the increasing and then decreasing rate of the rotating invaders in the outer circle.

THE PLAGUE OF CRABS

The crabs, sprite 33, are placed on the screen using PUTBLS.

The ROW and COL variables for the PUTBLS are loaded up each time by a random number produced by using the Forth word RND.

THE BOUNCING BALL

This is a sprite or to be more precise, 4 sprites, moving in background. The ball is defined in four orientations to give half character resolution. The movement is obtained by doing a PUTXRS, calculating the next co-ordinates, blotting out the old sprite with a second PUTXRS and then repeating the cycle. The listing for this screen is given in section 3 under the heading - Programmable Sprites.

THE LUNAR SPACE SHIP

The radar dish on top of the space ship, sprite 34, is animated by sequentially placing sprites 35 to 42, giving the effect of constant rotation.

Just before the spaceship takes off, an explosion, sprite 43, is XORed over the spaceship using PUTXRS. It is removed again by a second PUTXRS.

The spaceship is then scrolled up in a vertical window by 1 pixel. The Lunar surface, sprite 44, is also scrolled by 1 pixel, to the right using WRR1V.

THE ROTATING BALLS

Four sprites were used to give the impression of rotation, these being sprites 45, 46, 47, and 48.

Once the sprites were created, animation was a simple process of sequentially placing the sprites to the screen with an increasing and decreasing time delay between each PUTBLS.

THE CREDITS

To achieve the text scrolling up from the bottom of the screen, the bottom line was set with 0 INK and 0 PAPER colours, while the rest of the screen had 7 INK and 0 PAPER colours.

Each line of text was printed into that line and then the whole screen scrolled up 8 * 1 pixel, such that the text data scrolls into the screen that has 7 INK attributes and thus appears to smoothly scroll onto the screen.

SOME SIMPLE PROGRAMMING EXAMPLES

Example 1

To scroll a window at row 8, column 9, 5 characters high by 10 characters long, 1 pixel to the left with wrap - type:

```
: DEM1 5 HGT ! 10 LEN ! 8 ROW ! 9 COL ! WRL1V ; <CR>
```

To put some data on the screen: VLIST <CR>

To scroll the window 100 times type:

```
: DEM2 100 0 DO DEM1 LOOP ; <CR>
DEM2 <CR>
```

To execute DEM1 in background type: ' DEM1 INT-ON <CR>

To halt the background execution of DEM1 type: INT-OFF <CR>

Example 2

To scroll a window at column 12, row 3, 10 characters high and 5 characters wide, downward by 3 pixels with wrap - type:

```
: DEM3 10 HGT ! 5 LEN ! -3 NPX ! 2 ROW ! 12 COL ! WCRV ; <CR>
```

to define the word, then: DEM3 <CR> to execute it.

To run DEM3 in background type:

```
' DEM3 INT-ON <CR>
```

To halt the background execution type: INT-OFF <CR>

Example 3

To invert a window at column 10, row 5, 4 characters high and 4 characters wide - type:

```
: DEM4 4 HGT ! 4 LEN ! 5 ROW ! 10 COL ! INVV ; <CR>
```

and then DEM4 <CR> to execute.

Example 4

It is not always necessary to use colon definitions to achieve a particular result. To PUT sprite 34 (If you've got the Demo sprites loaded or have already defined your own sprite 34) at row 5, column 6, and replace any data currently at that position - type:

```
34 SPN ! 5 ROW ! 6 COL ! PUTBLS <CR>
```

Example 5

Often it is useful to define a word which carries out an operation that is used frequently and which saves typing - for instance:

```
: DEM5 COL ! ROW ! SPN ! PUTBLS ;
```

If we wanted to carry out the simple PUT command in example 4 all we'd need to type would be:

```
34 5 6 DEM5
```

Example 6

To scroll sprite 29 in memory by 1 pixel to the right with wrap and PUT it on the screen at row 10, column 11 - use:

```
: DEM6 29 SPN ! 10 ROW ! 11 COL ! WRR1M PUTBLS ;
```

To run DEM6 in background use: ' DEM6 INT-ON

Example 7

To fill a screen window, at column 17, row 5, 5 characters long and 10 characters high, with the attributes - 6 INK, 2 PAPER and 1 FLASH - use the following:

```
: DEM7 1 FLASH 6 INK 2 PAPER 10 HGT ! 5 LEN ! 13 ROW ! 17 COL ! SETAV 0 FLASH 7
INK 1 PAPER ;
```

Example 8

To change the green face of sprite 28 into a red one use:

```
: DEM8 28 SPN ! 2 INK ! 1 BRIGHT 0 PAPER SETAM PUTBLS ;
```

Example 9

To pick a random number between 0 and 100 use:

```
: DEM9 100 RND . ;
```

Example 10

To pick a random INK colour and change sprite 34's INK colour to this colour, and then put it at column 10, row 10, use:

```
: DEM10 7 RND INK 34 SPN ! SETAM 10 ROW ! 10 COL ! PUTBLS ;
```

To run DEM10 in background use: ' DEM10 INT-ON and to halt DEM10 use INT-OFF.

Example 11

To place sprite 24 into the screen of sprite 13 use:

```
: DEM11 24 SP1 ! 13 SP2 ! 7 SCOL ! 6 SROW ! GWBLM 10 ROW ! 10 COL ! ATTON PUTBLS
;
```

Example 12

To XOR sprite 43, the explosion, with sprite 27, the 270 degree rotated Invader, in memory, use:

```
: DEM12 43 SP1 ! 27 SP2 ! COPXRM 10 ROW ! 10 COL ! 27 SPN ! ATTON PUTBLS ;
```

Example 13

To invert sprite 7, the spider, in memory use:

```
: DEM13 7 SPN ! INVM 10 ROW ! 10 COL ! PUTBLS ;
```

Example 14

To enlarge sprite 24 into sprite 45 using the DSPM command use:

```
: DEM14 45 SP1 ! 24 SP2 ! DSPM 45 SPN ! 10 COL ! 10 ROW ! PUTBLS ;
```

Example 15

To search through sprite space, locate any existing sprites, print out the start of data, length and height, use:

```
: DEM15 255 1 DO I SPN ! TEST 1 = IF I. SPACE DPTR @ U. SPACE LEN ? SPACE HGT ?
SPACE CR THEN LOOP ;
```

Example 16

To scroll a landscape sprite numbered 128, 2 characters high and 64 characters wide, left by 1 pixel under interrupt, use the following:

```
SCR # 6
  0 0 VARIABLE CL 8 VARIABLE PH : OPEN EXX 0 COL ! 12 ROW ! 2 HGT !
  1 0 SROW ! 0 SCOL ! 128 SPN ! 0 PAPER 6 INK 32 LEN ! CLSV 0 INK
  2 1 LEN ! CLSV 32 LEN ! PWBLS EXX ;
  3 : NXB CL @ 1+ DUP 64 = IF DROP 0 ENDIF DUP CL ! 31 + 64 MOD SCOL
  4 ! 1 LEN ! PWBLS 32 LEN ! ;
  5 : SL WRL1V PH @ 1- DUP 0= IF NXB DROP 8 ENDIF PH ! ;
  6 : GO 6 INK 0 PAPER 0 BORDER 1 BRIGHT CLS 14 ROW ! 1 COL !
  7 6 PAPER 31 LEN ! 4 HGT ! SETAV OPEN 6 INK ' SL INT-ON ;
```

To speed this up to faster 4 or 8 pixel scrolls change lines 0 and 5 to read:

```
  0 0 VARIABLE CL 2 VARIABLE PH : OPEN EXX 0 COL ! 12 ROW ! 2 HGT !
  5 : SL WRL4V PH @ 1- DUP 0= IF NXB DROP 2 ENDIF PH ! ;

  0 0 VARIABLE CL 1 VARIABLE PH : OPEN EXX 0 COL ! 12 ROW ! 2 HGT !
  5 : SL WRL8V PH @ 1- DUP 0= IF NXB DROP 1 ENDIF PH ! ;
```

Type: 6 LOAD <CR> to compile and GO <CR> to run.

TRANSCRIBER'S NOTE

While OCRing and proofreading the scans of the White Lightning manuals, I have tried to preserve, as best as I was able, the original pagination, layout, spacing and formatting of the originals - while still incorporating all errata noted and published in previous versions, and with a few (small) edits and corrections of my own.

This transcription is still a work in progress. If you discover any further mistakes, please create a pull request (or otherwise report the errors) against the manuals' source repository at:

https://github.com/richmilne/white-lightning-manuals