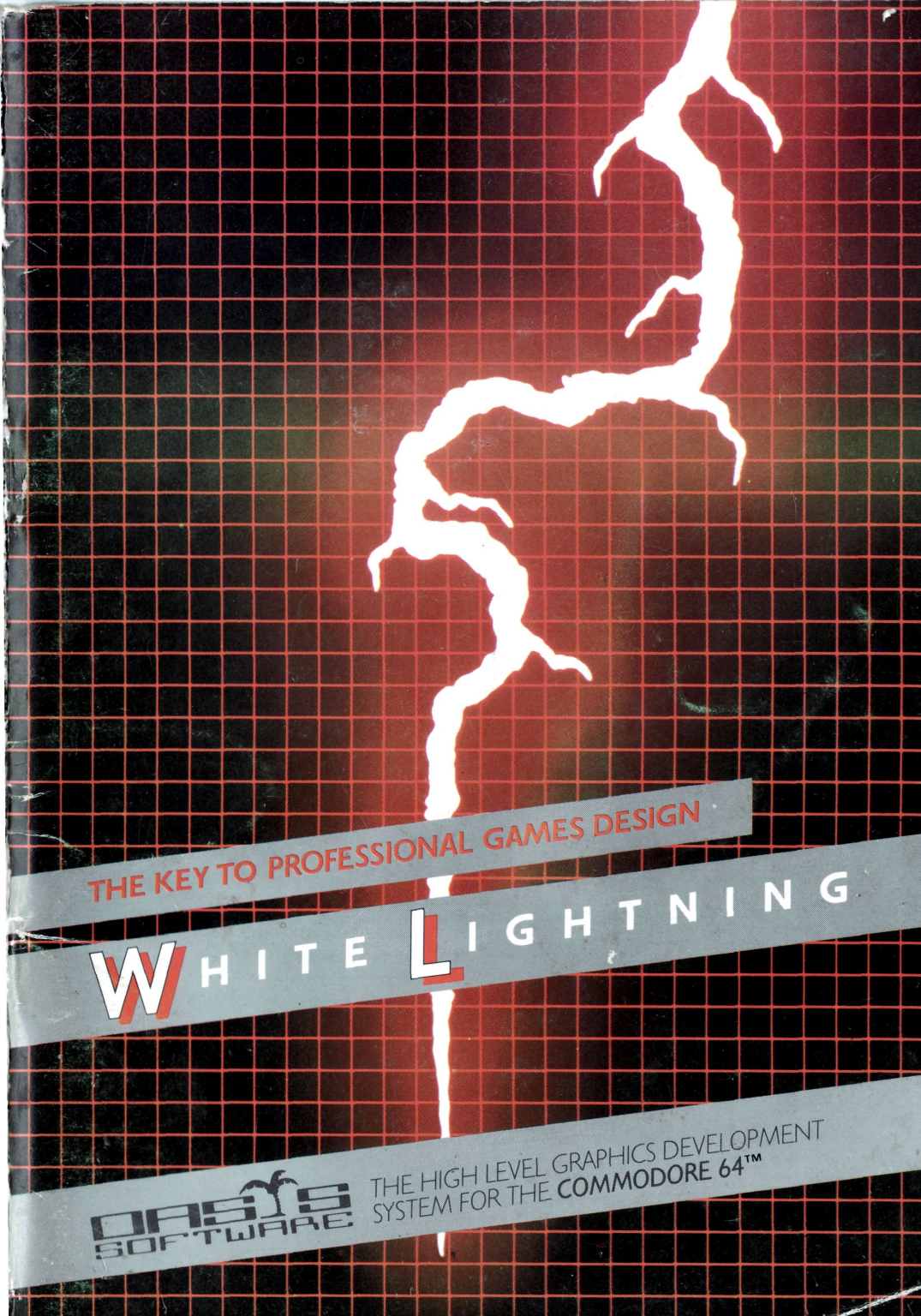

THE KEY TO PROFESSIONAL GAMES DESIGN
WHITE LIGHTNING
OASIS SOFTWARE
THE HIGH LEVEL GRAPHICS DEVELOPMENT SYSTEM FOR THE COMMODORE 64™

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.

NOTE

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

CONTENTS

	Page
INTRODUCTION	1
OPERATING INSTRUCTIONS	3
Loading White Lightning	3
Preparing a Disk for Source Code	3
Program Development	4
Saving a Finished Program	5
C64 FORTH	7
Input/Output Operators	8
Mathematical Operators	10
Stack Operators	13
Other Operations	14
Colon Definitions	15
Control Structures	16
Conditional Branching	17
Constants and Variables	21
Other Commonly Used Forth Words	21
Using the Editor	22
Editor Commands	23
Line Editor	23
Forth Error Messages	26
The Cassette Based System	27
The Disk Based System	28
Extending White Lightning	28
BASIC Interface	28
FIG-FORTH GLOSSARY	31
IDEAL GRAPHICS COMMANDS	61
Sprite Variables	61
Sprite Utilities	62
Display Modes	63
Setting the Attribute Value	63
PLOT, BOX, DRAW, POLY and POINT	64
Sprite Data Movement	66
Moving Attributes	69
Collision Detection (Software Sprites)	69
Clearing and Inverting Windows	70
Scrolling Commands	70
Transformations	72
Character Manipulation	73
READING THE KEYBOARD, JOYSTICK AND LIGHTPEN	74
Keyboard	74
Joystick	75
Lightpen	75
HARDWARE SPRITES AND SMOOTH SCROLLING	75
Defining and Hardware Sprite	75
Switching on a Hardware Sprite	76
Placing a Sprite on the Screen	77
Double-Sized Sprites	77

Multi-Coloured Sprites 77
Display Priorities 78
Hardware Sprite Collision Detection 78
Smooth Scrolling 78

SOUND COMMANDS 78
Volume 79
Frequency 79
Envelope (ADSR) 79
Waveform 79
Changing the Waveform 79
Filtering 81
Ring Modulation and Synchronisation 82
Examples 82/3
Mute, OSC and ENV 84

SPRITE STORAGE ORGANISATION 84

VARIABLE SETS 85

USING INTERRUPTS 85
PLAY 86
RPLAY 86
Format for Storing Tunes in Sprites 87
TRACK 87
MOVE 88

IDEAL GLOSSARY 89

IDEAL ERROR MESSAGES 101

WHITE LIGHTNING
by Oasis Software

INTRODUCTION

White Lightning is a high level graphics development system for the Commodore 64. 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 most 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.

White Lightning is Forth based and therefore 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.

At the time of writing there are two White Lightning User Groups. They are:

Mr T. Kelly,
White Lightning User Group,
353A Merville Garden Village,
Newtown Abbey,
Northern Ireland.

Mr M. Richards,
S.W. White Lightning User Group,
8 Victoria Road,
Roche,
Cornwall.

SPRITE DEVELOPMENT

Included with White Lightning is a sprite generator. This comes complete with a predefined arcade sprite set. You can use them as they are, customise them, or design up to 255 of your own sprites. The developement 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 or disk.

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.

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 60 times a second.

Extendible

Forth is extendible 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 in to the designing of the IDEAL animation language. The words were chosen to be as mnemonic as possible and their functions were selected to give as much power and flexibility as possible.

ACCESSING BASIC

If you are not familiar with Forth and want to produce reasonable software quickly, you can access the IDEAL language from BASIC. Programs will be less portable and you won't get quite the same speed and polish, but perfectly good programs can be, and have been, written in this way. More memory will be used for BASIC source, so bear this in mind before deciding to put off learning Forth!

OPERATING INSTRUCTIONS

To load White Lightning type:

```
SHIFT/RUN STOP          for tape users
LOAD"WL",8,1            for disk users
```

Once the program has loaded it will automatically do a COLD start and then go into White Lightning command mode.

To re-enter White Lightning from BASIC type:

```
SYS 4608                for a COLD start
SYS 4612                for a WARM start
```

Preparing a Disk

Users of the disk based White Lightning will need to prepare a disk for storing source code, sprites and semi-completed programs. To prepare a disk or disks use the following short BASIC program:

```
 5 REM FORMATTER
10 OPEN 15,8,15,"N0:name,id"
20 OPEN 5,8,5,"#"
30 FOR TR=1 TO 17
40 FOR SC=0 TO 20
50 PRINT#15,"B-A:0",TR,SC
60 NEXT SC
70 NEXT TR
80 CLOSE 15
90 CLOSE 5
100 END
```

This can be loaded from the White Lightning master disk using:

```
LOAD"FMAT",8
```

You should then place a blank disk in the drive and type RUN. Note that you can change line 10 to have your own name and id if you wish. This will format your disk and reserve sectors for your White Lightning source code. Don't forget to label your disk clearly.

The prepared disk now has:

```
357 sectors for White Lightning source code (approx 88 screens)
307 sectors for other files/programs        (approx 76k)
```

Remember that only specially prepared disks may be used for saving your source code on (this is done automatically by the editor when a FLUSH etc. is executed), but any disk may be used for sprites, semi-finished programs and ZAPped programs.

WARNING: Do not validate your specially prepared disk at any time or you will lose your source code.

Program Development

Source Code

Disk users can edit into screens 1 to 88 using the specially formatted disk described in the previous section. Don't forget to execute a FLUSH once your screen has been fully edited. This will send the updated screen to disk.

Tape users, however, have to keep their source code in RAM. Source code and sprites share memory between $6800 and $9900. This 12k or so of space is split with source code in the lower portion and sprites in the upper portion. In order to prevent source overwriting sprites or vice versa, the user must set the partition somewhere between the two using the LOMEM word.

```
HEX 6C00 LOMEM
```

would allocate 11k to sprites and only 1k (1 screen) to source. This would mean that the source would need to be loaded from tape and then compiled, one screen at a time.

```
HEX 7000 LOMEM
```

would allocate 10k to sprites and 2k (screens 1 and 2) to source code.

The values for LOMEM and the resulting memory allocations are summarised below:

LOMEM VALUE (HEX)	USABLE SCREEN	USABLE SPRITE SPACE
6C00	1	11K
7000	1- 2	10K
7400	1- 3	9K
7800	1- 4	8K
7C00	1- 5	7K
8000	1- 6	6K
8400	1- 7	5K
8800	1- 8	4K
8C00	1- 9	3K
9000	1-10	2K
9400	1-11	1K

If you are writing a large program it is best to set a high LOMEM value, load the source and compile it (if required, load a further batch of source into the same screen and continue compiling until all source is compiled). LOMEM can now be set as low as 6800 and sprites loaded into the 12k of space.

During the development of a White Lightning program it will be necessary to save:

1. The White Lightning nucleus and compiled dictionary.
2. The White Lightning source code screens.
3. The sprites.
4. The graphics routines.

The C64 cannot load from tape to addresses higher than $A000 and this is where the graphics routines reside. In order to overcome this problem White Lightning "packs" the routines down onto the White Lightning dictionary before SAVEing and relocates them back up after LOADing back in. This will corrupt source and sprites currently held in memory. As a result, SAVEing a semi-finished program requires the following procedure:

1. FLUSH the editing buffers (see section on editor).

2. Save sprites using " filename" STORE. (note the space between " and filename).

3. Save the source (tape version only) using: S1 S2 " filename" SCRSAVE. This saves from screen S1 to screen S2, to tape. So for instance, to save the source code for a game called "INVADERS" whose source code occupies screens 1 to 10, use:

   1 10 " INVADER " SCRSAVE

4. Save the semi-finished White Lightning program and graphics routines (this will corrupt the source and sprites currently held in memory) using: " filename" PACK. So to save "INVADERS" use:

   " INVADERS" PACK

   Note that when the SAVE is completed you are returned to White Lightning and the graphics routines are put back in their proper place. You will, however, need to re-load sprites and (tape version only) source code.

5. If you do wish to continue by re-loading sprites, use:

   " filename" RECALL

   to re-load sprites, and if you are using tape then use:

   " filename" BLKLOAD

   to re-load source code.

Tape users should note that if they have followed the above procedure and used a single tape then the three files should be in the order:

   Sprites, source code, White Lightning dictionary and graphics routines

Most users will probably find it easier to work with three labelled tapes.

It should also be noted that for most applications it is only necessary to save sprites and source code. The source code can be loaded back in as above and re-compiled, thus generating the White Lightning object code. PACK is normally used when producing your own extended version of White Lightning.

Saving a Completely Finished Program

Once you have completely finished and de-bugged the White Lightning program, developed all the sprites and successfully tested and compiled all the screens, you can now save the completed code in a form that will run without White Lightning present, (this final program must not return to command level as White Lightning is no longer present and would result in a crash), to do this you will need to ZAP your program.

ZAPping your program destroys certain areas of White Lightning, making it unusable as a program developer, but leaves intact that part responsible for program execution. The resulting program will be saved to tape or disk as a single program under the specified filename.

Before typing " filename" ZAP, be sure that:

1. You have compiled your program.

2. The last word in the dictionary is:

   HEX : nnnn 6800 LOMEM prog ;

   where "nnnn" can be any WORD and "prog" is the word which executes your final program.

3. All your sprites are present.

4. The loader program (see next section) has been saved to tape or disk.

5. Your tape or disk is ready to receive the final program.

6. Your program does not dynamically create sprites.

In order to ZAP your program all you need to type is:

```
" filename" ZAP
```

Note that the finished program could be up to 45k long, so if you are using a tape recorder a long tape and a fairly long wait could be required.

Reloading a ZAPped Program

The ZAPped program requires a small BASIC loader program to load and execute it. This should be saved to tape or disk before ZAPping your program and given the filename of the finished program.

Tape Version

```
20 POKE 183,0:POKE 184,1:POKE 185,1:POKE 186,1
30 FOR I=580 TO 591
40 READ B : POKE I,B
50 NEXT I
60 SYS 580
70 END
80 DATA 169,54,133,01,169,00
90 DATA 32,213,255,76,00,18
```

Disk Users

```
10 A=A+1 : IF A=1 THEN LOAD "filename",8,1
20 SYS 4608
```

The "filename" in line 10 is once again the name of the ZAPped file.

So, your final program consists of a BASIC loader and a machine code program and all that is required to run your final program is to load and run the BASIC loader. Happy Zapping!

C64 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 execute lines of BASIC from within Forth itself. Only the standard Commodore BASIC commands can be used in this manner but the aim of this extension is to provide the newcomer to Forth with a gradual transition. The second and most important addition is the IDEAL 64 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 64 editor can be used to create lines of White Lightning source code for later compilation. Do not allow lines to exceed 63 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 $6800 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,PO BOX 1105, SAN CARLOS, CA 94070).

C64 Forth was written by Stuart Smith, the author of the extremely successful DRAGONFORTH and SPECTRAFORTH, and is an enhancement of a program written by the Forth Interest Group - to whom we offer our thanks.

If you are using a tape based version instructions referring to discs should be interpreted as accessing RAM. The disk version does not require any RAM to store screens in as it writes any updated screens to disk.

AN INTRODUCTION TO C64 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 must 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 n LOAD <CR> (where n is the number of the screen to be compiled). Throughout these examples <CR> means 'PRESS RETURN'.

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, C64 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 may be any of the recognisable ASCII characers, a graphic character, or a control code depending on the value of the number on the stack.

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.

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 on 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 6000     TOS
            FFEE
```

This will store FFEE at address 6000/6001
i.e. EE at 6000
FF at 6001

If we key in HEX FF00 6000 ! <CR>
this will store FF00 at 6000/6001
i.e 6000 contains low byte 00
6001 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 "Fetch")

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 6000 C! <CR>
will store a single byte FF at address 6000.

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 6000 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 6000 +! <CR>
will add 4 to the value at 6000/6001
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. Use of the 64 full screen editor is also allowed.

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**

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 DO ." 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	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 ... ENDCASE

For example type:

: TEST4 CASE 1 OF ." FIRST CASE " ENDOF 2 OF ." SECOND CASE " ENDOF 3 OF ." THIRD CASE " ENDOF ENDCASE ; <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 Forth 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. Pressing run/stop will stop the listing.

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 run/stop 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 become obvious in the next section on editing.

USING THE EDITOR

To begin with, whilst you are getting used to this package, most of your use will probably be in immediate mode. Clearly if you are going to write full programs you need to be able to see what you've written so far, and have a facility to make minor changes without typing the whole lot in again.

Forth Source

Forth is not an interpreted language like BASIC and it's probably worth a brief explanation of the differences before going much further.

Interpreted language programs consist of lines of text (source code) which are read by the interpreter and then directly executed. The source code is the program. Forth, and for that matter most other high level languages as well as assembly language, do not directly execute the source code. When you type N LOAD (where N is the screen at which compilation will begin) the Forth compiler reads the source code in that screen and translates it into what amounts to a series of machine code calls. These calls (with a few exceptions which do not concern us here) are not executed, however. The final program is executed by typing in one of the words defined in the source code.

Writing a BASIC program usually consists of typing a line number followed by the text of the program that makes up that line. In this way, lines can be inserted and deleted in the order required by the user. A BASIC program is one continuous block of text.

Forth source, on the other hand, is not one continuous block of text, but is divided into screens which are themselves divided into 16 lines of 63 characters. The program is built up by a line at a time in each of the screens. The first step is to pick a current screen to work on. Let's suppose we are writing a program that starts at screen 1. To select screen 1 we could type either 1 CLEAR or 1 LIST. If we typed 1 CLEAR the screen would be cleared of all text data so this is what we do the first time we use it (this gets rid of any garbage). Once we have begun work on a screen then it is selected by using 1 LIST. This will display the screen and select it as the current screen so that the EDIT command will now refer to lines within that screen. Now that we've selected screen 1 all we have to do is select a particular line to edit. Let's edit line 4. To do this type 4 EDIT and

```
4 P
```

will appear on the screen. Now use the line editor as normal to insert the text

```
4 P : FRED ." THIS IS FORTH " ;
```

Now type 1 LIST and the screen will list with your new line 4 included. Now type 1 LOAD to compile this source code. Now type FRED to execute your new word and you should see "THIS IS FORTH" printed on the screen.

Now let's look at the editor commands in more detail.

EDITOR COMMANDS

As previously discussed, before using a particular screen for the first time, it is necessary to CLEAR it of any extraneous data. To do this, simply key in:

```
n CLEAR
```

where n is the number of the screen you wish to clear. If you clear a screen that already contained data, then it will be wiped out, so be careful. You may now enter new text or change existing text on any line of the current screen by using the word EDIT. EDIT simply prints out the line number desired, followed by the letter P, followed by any existing text, for example:

```
5 EDIT
5 P THIS IS LINE FIVE
```

You may now use the full screen editor to change the text on the line and then press RETURN. Note that any characters after the 63rd character are 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 1024 bytes, divided into 16 lines of 64 characters, of which 63 characters are used. 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.

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 the 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 : Allows use of the 64 full screen editor. Also, the cassette version 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. Note that the disk version does NOT do an automatic FLUSH.

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. 7 LIST will now make screen 7 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 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, or to the disk drives. To do this, just key in

FLUSH <CR>

Note that the EDIT command does an automatic FLUSH only for the cassette version. You can save your text to tape using the BLKSAVE command.

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:

5 CLEAR <CR>

This sets the current screen to 5. 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 P THIS IS HOW TO PUT <CR>
1 P TEXT ON LINE 1 <CR>
2 P LINE 2 <CR>
3 P AND LINE 3 OF THIS SCREEN <CR>
```

5 LIST will produce:

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

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 5 LIST <CR> to see the result. The editor ignores characters after the 63rd character of the line being edited.

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 may occur when editing, loading or listing screens of data.

Try: 1000 LIST <CR>

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

7 - this indicates dictionary full.

8 - Device I/O error.

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

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-existant line of screen data.

Try: 12 D

The Cassette Based System:

EDIT does an automatic FLUSH and LIST.

At the end of an editing session you will require to save your source code to tape. To do this:

S1 S2 " name" SCRSAVE

where S1 = first screen to be saved
S2 = last screen to be saved
name = name of cassette file.

To LOAD your screens back in from tape, use:

EMPTY-BUFFERS
" name" BLKLOAD

Note that the EMPTY-BUFFERS will clear out the editing buffer before the cassette load. If you do NOT do this then some garbage may be present.

The Disk Based System:

The disk based White Lightning has a number of advantages over the tape based system. The ease and speed of loading from disk make for a much more rapid development cycle and the extra storage capacity means that larger programs can be written. There are some differences which should be noted:

Unlike the cassette version, where screens are RAM based, the disk based version does not automatically execute a FLUSH after each EDIT and so, once a screen has been fully edited, it is necessary to execute a FLUSH to update the disk source code. There is no need to use SCRSAVE or BLKLOAD.

From time to time it is possible that a disk may have a bad sector. This problem can be quite easily circumvented by missing out the offending screen.

Each sector on the disk is made up of 256 bytes, so there are four sectors per Forth screen. Forth screens are mapped directly to disk sectors. Before using a screen for the first time it is always advisable to CLEAR it first to remove any garbage. If this new screen does contain a bad sector then Forth will issue error message 8. Let's consider an example.

Suppose your program is four screens long and you intend to use screens 2 to 5 inclusive to hoold your program. Normally the last instruction of screens 2 to 4 would be --> (minus, minus, greater than) to tell Forth to continue loading at the next screen. Suppose, however, that screen 4 was found to contain a bad sector. This means that the code will now be edited into screens 2, 3, 5 and 6 and we need to jump over screen 4. All that is required is to change the --> at the end of screen 3 to become 5 LOAD. This will instruct Forth to continue LOADing at screen 5 instead of simply continuing to screen 4.

BASIC INTERFACE

As well as the large number of Forth graphics and sound commands available with White Lightning, there is also the ability to access most of the BASIC commands as well (see list further on for limitations). Only Commodore BASIC can be accessed in this way.

The BASIC statements can be accessed in the following manner:

```
B[ statement 1 : statement 2 :]
```

e.g. `B[PRINT "HELLO" : PRINT "BYE BYE"]`

Note that multiple statements may be put between B[and]

Any errors encountered during BASIC will lead to normal error messages being printed and Forth will do a WARM START.

It is possible to execute BASIC commands immediately or to place them within colon definitions:

```
B[ PRINT "HELLO"]
```

will immediately execute and print HELLO.

```
: TEST B[ PRINT "HELLO"] ;
```

will compile a word called TEST which can be executed by keying in

```
TEST
```

which will again print HELLO.

When handling strings in BASIC via B it is necessary to take the following steps if handling strings in the 'immediate' mode, i.e. outside colon definitions:

The string must be defined with a NULL string added in order to preserve it. For example, try:

```
B[ A$= "WHITE LIGHTNING"]
B[ PRINT A$]
```

and you will see that the string has been lost. Now try:

```
B[ A$= "WHITE LIGHTNING"+""]
B[ PRINT A$]
```

and you will see that this time the string has been preserved.

The addition of the NULL string is only necessary if using strings outside colon definitions. Try:

```
: STRING B[ A$= "WHITE LIGHTNING"] ;
  STRING
  B[ PRINT A$]
```

and the string is OK.

LIMITATIONS

In general, any commands which involve line numbers, or any commands which cannot be used in the immediate mode, may NOT be used in White Lightning.

The following commands require line numbers and therefore cannot be used:

GOTO
GOSUB
ON ... GOTO
ON ... GOSUB
RETURN

The following commands generate errors when used in the immediate mode and therefore cannot be used:

INPUT
DEF
READ
DATA

The following commands require a complete BASIC program to work on and cannot be used:

LIST
CONT
NEW
RUN

DIM should be used with great care as there is only limited space available for variables within BASIC - approximately 3k to be shared between numeric and string variables.

Forth words:

B[
(B)
(TO)
(EX)

are all used as part of the BASIC interface.

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.
LO	Level zero definition of FORTH-78.
LI	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: For the cassette based system, all references to disc in this documentation can be read as references to the disc simulation area in memory from 6000H upwards, which is treated as a very limited disc capacity by White Lightning, and does not in any way change the operation or description of any of the FORTH words defined in this documentation.

! n addr --- **LO**

Store 16 bits of n at address. Pronounced "store".

!CSP

Save the stack position in CSP. Used as part of the compiler security.

dl --- d2 LO

Generate from a double number dl, the next ASCII charater 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 LO

Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#S dl --- d2 LO

Generates ascii text in the text output buffer, by the use of #, until a zero double number results. Used between <# and #>.

' --- addr P,LO

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,LO

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.

(OPEN)

Used in the form:
file no. device no. sec.addr "filename" OPEN
to open a file for input/output.

(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 **LO**

Leave the signed product of two signed numbers.

*/ n1 n2 n3 --- n4 **LO**

Leave the ratio of 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 LO

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 LO

Leave the sum of n1+n2.

+! n addr --- LO

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 addr1 --- 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,LO

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 --- LO

Store n into the next available dictionary memory cell, advancing the dictionary pointer. (comma).

- nl n2 --- diff LO

Leave the difference of nl-n2.

--> P,LO

Continue interpretation with the next screen. (Pronounced next-screen).

-DUP nl --- nl (if zero)
  nl --- nl nl (non-zero) LO

Reproduce nl 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 nl --- addr n2

Adjusts the character count nl of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+nl to addr+n2 are blanks.

. n --- LO

Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing block follows. Pronounced "dot".

." P,LO

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 nl n2 ---

Print the number nl right aligned in a field whose width is n2. No following blanks printed.

/ n1 n2 --- quot LO

Leave the signed quotient of n1/n2.

/MOD n1 n2 --- rem quot LO

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 LO

Leave the true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n --- f LO

Leave a true flag if 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.

2DROP d1 ---

Drop double precision number from top of stack or drop two single precision numbers.

2DUP n2 n1 --- n2 n1 n2 n1

Duplicates the top two values on the stack. Equivalent to OVER OVER.

: P,E,LO

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,LO

Terminate a colon-definition and stop further compilation. Compiles the run-time ;S

;CODE P,C,LO

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 6502 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,LO

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 LO

Leave a true flag if n1 is less than n2; otherwise leave a false flag.

<# LO

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<LO

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 be written in high level, rather than in assembler code (as required by ;CODE).

= n1 n2 --- f LO

Leave a true flag if n1=n2 otherwise leave a false flag.

> n1 n2 --- f LO

Leave a true flag if n1 is greater than n2 otherwise leave a false flag.

>R n --- C,LO

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 --- LO

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.

?DUP

Same as -DUP.

?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 nl n2 ---

Issue an error message if nl does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

Issue an error message if the stack is out of bounds.

?TERMINAL --- f

Perform a test of the terminal keyboard for actuation of the run/stop key. A true flag indicates actuation.

@ addr --- n LO

Leave the 16 bit contents of address.

ABORT LO

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 LO

Leave the absolute value of n as u.

AGAIN addr n --- (compiling) P,C2,LO

Used in 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 --- LO

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 nl n2 --- n2 LO

Leave the bitwise logical "AND" of nl 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 1024 bytes, organised as 16 lines of 64 characters each (63 of these are usable).

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,LO

Occurs in a colon-definition in the form:

```
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetetively 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.

A 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 LO

A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLKLOAD addr ---

Load contents of RAM from device using filename stored at addr (usually PAD). Note that the load address is the same as the save address used when saving the file.

Used in the form: " name" BLKLOAD

BLKSAVE addr addr addr ---

Save contents of RAM from addr to addr using the filename stored at addr (usually PAD). This must be used in the form:

A1 A2 " name" BLKSAVE

BLOCK n --- addr LO

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,LO

The run-time procedure for 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 rea- 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.

C/L --- n

A constant containing the number of characters per line (64).

CASE --- n (compiling)

Occurs in a colon definition in the form:

```
CASE
n OF ..... 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.

CHKIN channel ---

Open a channel for input. OPEN must previously have been called.

CHKOUT channel ---

Open a channel for output. Must previously have called OPEN.

CHRIN --- char

Get a character from current input channel. Must previously call OPEN and CHKIN unless using keyboard.

CHROUT char ---

Output the character at the top of the stack to the current output channel. Previously call OPEN and CHKOUT unless using the screen.

CLOSE file no. ---

Close a logical file number which is on top of the stack.

CLRCHN ---

Clear the input/output channels and restore them to their default values, i.e. keyboard and screen.

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 towards 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 --- LO

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 onto the stack.

CONTEXT --- addr U,LO

A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr 1 --- addr 2 LO

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 LO

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 --- LI

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 --- DO

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 LO

Set the numeric conversion BASE for decimal input-output.

DEFINITIONS LI

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 nl --- n2 tf (ok)
c nl --- ff (bad)

Converts the ASCII characters c (using base nl) 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 dl --- d2

Convert dl to its double number two's complement.

DO nl n2 --- (execute)
addr n --- (compile) P,C2,LO

Occurs in a colon-definition in the form:

```
DO ... LOOP
DO ... +LOOP
```

At run time, DO begins a sequence with repetetive 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> LO

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,LO

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 --- LO

Drop the number from the stack.

DUMP addr --- LO

Print the contents of 80H memory locations beginning at addr. Both addresses and contents are shown in the current numeric base.

DUP n --- n n LO

Duplicate the value on the stack.

ELSE addr1 n1 --- addr2 n2
(compiling) P,C2,LO

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 --- LO

Transmit ASCII character c to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS LO

Make 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,LO

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,CO,LO

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 compiletime, 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 test of line n, relative to screen 4 and 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 --- LO

Transfer characters from the terminal to address, until a return has been received. One or more nulls are added at the end of the text. Note RETURN must be pressed before the count of characters has been reached. Full use of the screen editor is available.

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 the 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,LO

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,LI

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 LO

Leave the address of the next available dictionary location.

HEX LO

Set the numeric conversion base to sixteen (hexadecimal).

HLD --- addr LO

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c --- LO

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,LO

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,LO

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 OBRANCH 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 LO

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 "7" 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.

JSR addr ---

Call the machine code subroutine at addr.

KEY --- cc LO

Leave the ASCII value of the next terminal key struck.

L ---

List the current screen.

LATEST --- addr

Leave the name field address of the topmost word in the current vocabulary.

LEAVE C,LO

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 --- LO

Display the ASCII text of screen n on the selected output device. SCR contains the screen number during and after this process. Pressing run/stop will stop the listing.

LIT --- n C,LO

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,LO

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 --- LO

Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->.

LOMEM addr ---

Set the upper limit for tape source, which is the lower limit for sprite space, to the value addr.

LOOP addr n --- (compiling) P,C2,LO

Occurs in a colon-definition in the 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 LO

Leaves the greater of two numbers.

MERGE addr ---

Merge sprites onto the end of current sprites using the filename stored at addr. Used in the form:

" name" MERGE

No check can be made as to the length of the file being merged so exercise care to ensure the graphics routines above sprite space are not overwritten.

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 **LO**

Leave the smaller of two numbers.

MINUS n1 --- n2 **LO**

Leave the two's complement of a number.

MOD n1 n2 --- mod **LO**

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.

NOOP ---

This will perform a no-operation, i.e. do nothing.

NUMBER addr --- d

Convert a character string left at addr with a preceeding 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.

OPEN

Used in the form:

file no. device no. sec.addr. " filename" OPEN

to open a file for input/output.

OR nl n2 --- or LO

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 nl n2 --- nl n2 nl LO

Copy the second stack value, placing it as the new top.

PACK addr ---

Pack graphics routines down onto the dictionary using filename stored at addr. Used in the form:

" name" PACK

This will overwrite and possibly corrupt sprites or source code and is only rarely used during program development. Sprites and source (in the case of the tape version) should be re-loaded from tape/disk before continuing the session.

PAD --- addr LO

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 a 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

Gets up to 80 characters of text from keyboard. Allows full use of the 64 screen editor. RETURN must be pressed before the 81st character is input. Text is positioned at the address contained in TIB with IN set to zero.

QUIT LI

Clear the return stack, stop compilation and return control to the operator's terminal. No message is given.

R --- n U

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 ---

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 LO

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!

RECALL addr ---

Load sprites from the filename stored at addr. Used in the form:

" name" RECALL

This will overwrite any sprites previously stored in memory.

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.

RND n1 --- n2

Leave the random number n2 in the range 0 to n1.

ROT n1 n2 n3 --- n2 n3 n1 LO

Rotate the top three values on the stack, bringing the third to the top.

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 referenced by LIST.

SCRSAVE S1 S2 addr1 ---

Used to save contents of screens to device.

S1 = first screen to be saved
S2 = last screen to be saved
addr1 = address of filename preceded by byte count

Used in the form:

S1 S2 " name" SCRSAVE

Note that users of the disk based system will not need to use this word.

SETLFS file no. device no. sec.addr. ---

Set up logical file number, device address and secondary address.

SETNAM addr ---

Set up the filename. Assumes use in the form:

" name" SETNAM

SIGN n d --- d LO

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 initialise the stack pointer from S0.

SP@ --- addr

A computor 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 --- LO

Transmit n ASCII blanks to the output device.

STATE --- addr LO,U

A user variable containing the compilation state. A non-zero indicates compilation. The value itself may be implementation dependent.

STORE addr ---

Save sprites using the filename stored at addr. Used in the form:

" name" STORE

SWAP n1 n2 --- n2 n1 LO

Exchange the top two values on the stack.

TEXT c ---

Accept the following text to PAD. c is the text delimiter.

THEN P,CO,LO

An alias for ENDIF.

TIB --- addr U

A user variable containing the addresses of the terminal input buffer.

TOGGLE addr b ---

Complement the contents of addr by the bit pattern b.

TRAVERSE addrl n --- addr2

Move across the name field of a Fig-FORTH variable length name field. addrl 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 --- LO

Transmit count characters from addr to the selected output device.

U< ul u2 --- f

Leave the boolean value of an unsigned less-than comparison. Leaves f=1 for ul > u2; otherwise leaves 0. This function should be used when comparing memory addresses.

U* ul 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 ul --- u2 u3

Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor ul.

U/MOD

Same as U/

UNTIL f --- (run-time)
addr n --- (compile) P,C2,LO

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 LO

Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred to disc should it's 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 --- LO

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 it's offset and the user base address on the stack, as the storage address of that particular variable.

VARIABLE E,LO

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 "run/stop" will terminate the listing.

WARM ---

This will perform a warm-start.

WARM->COLD

This allows you to preserve any Forth word defined to date, so that a COLD start will not delete them.

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 nl 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 nl --- addr1 nl 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 --- LO

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 occurances 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 LI

Leave the bit-wise logical Exclusive-OR of two values.

ZAP addr ---

Produce a packed stand alone program using the filename stored at addr. Used in the form:

" name" ZAP

P,LI

[
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.

LI

]
Resume compilation, to the completion of a colon-definition. See [

IDEAL GRAPHICS AND SOUND COMMANDS

By David Hunter

The graphics commands included in WHITE LIGHTNING are designed to allow easy manipulation of images on and off the screen. This is achieved by carrying out operations on a table of sprites which can initially hold up to 6k of data although this can easily be changed. In this context, "sprite" means a graphics character of user-definable dimensions (up to 255 characters height or width) which may be displayed on the screen in one of the high-resolution modes (either bit-map mode or multi-colour bit-map mode). This is not to be confused with the 64's own hardware sprites - these can also be used from WHITE LIGHTNING and will be dealt with later.

Each sprite in the table is given a number from 1 to 255, and the screen is treated as sprite number zero - thus, the same commands can be used for both the screen and the sprites.

Note that <CR> means "Press RETURN" and should not actually be typed.

SPRITE VARIABLES

The sprite graphics commands in WHITE LIGHTNING use fifteen variables to pass parameters. The variables are:

SPN	sprite number 1
COL	column in sprite number 1
ROW	row in sprite number 1
WID	width of window
HGT	height of window
SPN2	sprite number 2
COL2	column in sprite number 2
ROW2	row in sprite number 2
NUM	number of sides on polygon or number of pixels to scroll
INC	inclination of polygon
ATR	current attribute
CCOL	column for collision detection
CROW	row for collision detection
SPST	start of sprite storage
SPND	end of sprite storage
SET	current variable set (not a true variable)

(The way that these variables are used will become clear later).

They can be treated like normal FORTH variables; i.e. they can be assigned a value:

```
3 SPN  !
```

or have their values fetched:

```
SPN @ .
```

The first sprite words that we will look at are CLR, ISPRITE and DSPRITE, which are used to create and delete sprites in the table.

CLR has no parameters, and it simply removes all the sprites from the table.

ISPRITE creates a new sprite in the table with number SPN, width WID and height HGT character blocks. All the data in the sprite is cleared when it is created. For example, 1 SPN ! 16 HGT ! 16 WID ! ISPRITE would create space for a sprite number 1, 16 character blocks square. Each character block in the sprite takes up ten bytes - eight bytes for the pixel data, one byte for the primary attribute data and one byte for the secondary attribute data. (The attributes determine what colour the pixel data will be displayed in - only the primary attributes are used when in two-colour mode). Also, there is an overhead of seven bytes for each sprite.

DSPRITE will remove sprite no. SPN from the table, releasing the space for new sprites. Note that trying to delete a sprite which doesn't exist will give an error message, and you are not allowed to delete sprite no. zero, as the screen is treated as sprite zero. Also, you cannot create a new sprite using ISPRITE if a sprite with that number already exists.

DFA, AFA and AFA2 leave on the stack the starting addresses of the pixel data, the primary attribute data and the secondary attribute data of sprite SPN respectively. They also set WID and HGT to the size of the sprite. If a sprite is undefined, they return values of -1. So a word to delete a sprite without giving an error if it doesn't exist is:

```
: DELETE DFA 1+ IF DSPRITE THEN ;
```

Since the hi-res screen is "hidden" under the KERNAL ROM, you cannot read data from it using @ (fetch).

SAVING and LOADING Sprites

The word STORE saves the sprites currently in memory to tape or disk. It is similar to BLKSAVE, the difference being that the start and end addresses do not have to be specified:

```
" filename" STORE
```

There are two words which can be used to load sprites into memory: RECALL and MERGE. RECALL will overwrite any sprites that are already in memory, while MERGE will add the new sprites onto the sprites that are already there. Both have the same syntax as STORE:

```
" filename" RECALL
" filename" MERGE
```

Note that there is nothing to stop you from loading in more sprite data than there is room for in memory - this will overwrite the graphics routines and could crash the system, so be careful!

If you have just MERGEd more sprites onto those in memory, it is possible that more than one sprite will have the same number. You can renumber all the sprites using RESEQ which renumbers them from 1 in steps of 1. If there is not enough room for the new sprites being MERGEd into memory, the sprites already present will be removed.

DISPLAY MODES

Data can be displayed on the hi-res screen in either two-colour mode or four-colour mode. In two-colour mode, each character block contains 64 (8x8) pixels. In four-colour mode, each pixel can take on one of four colours, but each character block contains only 32 pixels, since each pixel is twice as wide as in two colour-mode. MONO and MULTI put the hardware into two-colour and four-colour modes respectively.

S2COL and S4COL govern whether the sprite commands operate on two-colour data or four-colour (multi-colour mode) data. Thus it is possible to display a picture on the screen in two-colour mode while preparing data to be displayed in four-colour mode.

SETTING THE ATTRIBUTE VALUE

SETATR is a word which sets up the value of ATR, the current attribute value. When in S2COL mode, it takes the form: background foreground 0 SETATR. "foreground" and "background" are one of the following:

BLACK	WHITE	RED	CYAN
PURPLE	GREEN	BLUE	YELLOW
ORANGE	BROWN	.RED	GRAY1
GRAY2	.GREEN	.BLUE	GRAY3

(These are in fact predefined constants which return values in the range 0 to 15).

A full stop before the colour should be read 'light' - e.g. ".GREEN" is light green.

In S4COL mode, use:

colour2 colour1 colour3 SETATR (note the order!)

(Colour zero is the same for the whole screen).

PAPER, BORDER AND INK COLOURS

TPAPER, TBORDER, HPAPER and HBORDER define the paper (background) and border colours used in LORES (TEXT) mode and HIRES mode. For example:

BROWN TPAPER YELLOW TBORDER

will give a brown background with a yellow border when in text mode, and

BLACK HBORDER

will give a black border when in hi-res mode.

Also 'colour INK' will set the colour used when printing characters in text mode.

The word LORES puts the screen into text mode, while the word HIRES puts it into hi-res mode. To put a text window on the screen, use n WINDOW which will make the top n lines of the screen hi-res, and the rest text. For example, 16 WINDOW will give you 16 lines of hi-res at the top of the screen and 9 lines of text at the bottom. When a window is set up, the hi-res border colour is used. Note also that use of a disk drive while a window is set up will make it flicker.

Now we will look at some words which place data inside sprites:

PLOT, BOX, DRAW, POLY and POINT

First type 16 WINDOW to set up a screen window. Unless you have already used the computer, the upper part of the screen will be filled with garbage which must be removed.

SCLR clears all the pixel data in sprite SPN and sets the attributes to ATR. Type "ATTON" (this is explained later) and then WHITE BLACK 0 SETATR to set up the attribute used. Now, if you type 0 SPN ! SCLR the upper part of the screen will be cleared. Note that an SCLR is done automatically when you create a sprite using ISPRITE.

PLOT is used to set or clear individual points in a sprite; its parameters are SPN, COL and ROW.

If you type 0 COL ! 0 ROW ! PLOT you will see that the point at the top left corner of the screen is set. Try using other values of COL and ROW. The maximum value of COL is 319, while the maximum value of ROW is 199, although you cannot see any points plotted with ROW greater than 127 because you have set up a text window.

PLOT can also be used to clear points in a sprite to the background colour or to toggle (invert) a point; this is achieved by using the word MODE first. 0 MODE or 1 MODE will cause points to be set to background colour, while 2 MODE or 3 MODE will cause them to be set to the foreground colour. If you use 4 MODE, the points will be inverted.

In S4COL mode, 0, 1, 2 and 3 correspond to the background colour, colour 1, colour 2 and colour 3 (see the section "SETTING THE ATTRIBUTE VALUE"), and mode 4 will cause colour 3 to change to background, colour 2 to change to colour 1 and vice versa. For example, if you now type 0 MODE PLOT, the last point that you set will be cleared. The following word will make it flash on and off:

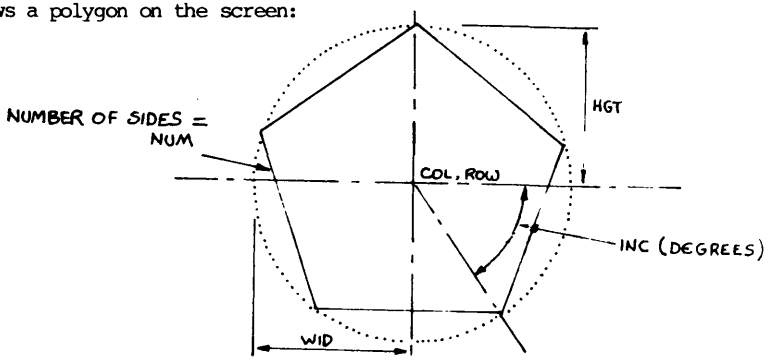
```
: FLASH 4 MODE 0 SPN ! 0 ROW ! 0 COL ! 100 0 DO PLOT 1000 0 DO LOOP LOOP ;
```

MODE also affects the BOX, DRAW and POLY commands.

DRAW draws a line inside sprite SPN from the point (COL,ROW) to (COL2,ROW2). For example, 0 COL ! 0 ROW ! 319 COL2 ! 127 ROW2 ! DRAW will draw a diagonal line across the screen.

BOX plots a rectangular block inside a sprite. The top left corner is at COL,ROW. WID is the width and HGT is the height of the block. 3 MODE 150 COL ! 54 ROW ! 20 WID ! 20 HGT ! BOX will draw a 20x20 pixel square at the centre of the screen.

POLY draws a polygon on the screen:



COL and ROW are the centre of the polygon. WID is the horizontal radius and HGT is the vertical radius. INC is the inclination in degrees and NUM is the number of sides. If NUM is large or less than three, a circle (or ellipse) is drawn instead of a polygon. For example, 32 COL ! 32 ROW ! 32 WID ! 32 HGT ! 0 SPN ! 5 NUM ! 0 INC ! POLY draws a pentagon in the top left of the screen. 36 INC ! POLY draws another one over it, while 0 NUM ! POLY draws a circle.

Here is a program which illustrates the use of DRAW and POLY:

```
SCR#1
 0 ( POLY AND DRAW EXAMPLE)
 1
 2 0 VARIABLE QUADADDR : QUAD QUADADDR @ EXECUTE ;
 3 : EX3 .GREEN BLACK 0 SETATR S2COL 0 SPN ! SCLR
 4 3 MODE MONO HIRES 100 160 48 QUAD ;
 5 : IQUAD DUP 2 > IF >R DUP COL ! OVER ROW ! R 2 * 3 /
 6 DUP WID ! HGT ! 0 INC ! 0 NUM ! POLY
 7 OVER R + OVER R 2  /  QUAD
 8 OVER OVER R + R 2  /  QUAD
 9 OVER R - OVER R 2  /  QUAD
10 OVER OVER R - R 2  /  QUAD
11 OVER R - ROW ! DUP COL ! OVER R + ROW2 ! DUP COL2 ! DRAW
12 OVER ROW ! DUP R - COL ! OVER ROW2 ! DUP R + COL2 ! DRAW
13 R> THEN DROP DROP DROP ;
14 ' IQUAD CFA QUADADDR !
15 ;S
```

After compiling the screen, type EX3 to draw an intricate pattern on the hires screen.

Notice that IQUAD in line 5 is recursive; it calls itself.

POINT is used to examine a pixel on the hi-res screen, and has parameters SPN,COL,ROW. In S2COL mode, it leaves on the stack a value of 0 or 1, corresponding to the point being cleared or set respectively. In S4COL mode, it returns 0,1,2 or 3 corresponding to background or the three colours.

Although the pixels are twice as wide in multi-colour mode, the scaling when using COL is still the same; in S4COL mode, COL=0 and COL=1 will refer to the same pixel.

SPRITE DATA MOVEMENT

WHITE LIGHTNING includes 39 words for moving rectangular blocks of sprite data:

MOVBLK	PUTBLK	GETBLK	CPYBLK
MOVOR	PUTOR	GETOR	CPYOR
MOVXOR	PUTXOR	GETXOR	CPYXOR
MOVAND	PUTAND	GETAND	CPYAND
BLK%BLK	OR%BLK	XOR%BLK	AND%BLK
BLK%OR	OR%OR	XOR%OR	AND%OR
BLK%XOR	OR%XOR	XOR%XOR	AND%XOR
BLK%AND	OR%AND	XOR%AND	AND%AND
MOVATT	SWAPATT		
ATTOFF	ATTON	ATT2ON	
DTCTON	DTCTOFF		

Note that the screen is treated throughout as Sprite 0 with height 25 characters and width 40 characters.

ATTOFF,ATTON and ATT2ON control the movement of attribute data with the pixel data. After executing ATTOFF, attribute data movement is disabled. Only the primary attribute data is moved after ATTON, and both sets (primary and secondary data) are moved after ATT2ON. The screen's secondary attribute data uses the same memory as the text colour memory, so you must not scroll the text screen if you are displaying data in multi-colour mode.

Thus, to go into two-colour mode, use

```
MONO S2COL ATTON
```

and for four-colour mode, use

```
MULTI S4COL ATT2ON
```

ONE-WAY DATA MOVEMENT.

All the data movement words have been named to make them easy to remember:

SUFFIXES:

BLK	the sprite data overwrites its destination.
OR	the sprite data is ORed with its destination.
AND	the sprite data is ANDed with its destination.
XOR	the sprite data is exclusive-ORed with its destination.

PREFIXES:

MOV parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
A window of width WID and height HGT in sprite SPN whose top left-hand corner is at COL,ROW is MOVed into SPN2 at COL2 and ROW2.

PUT parameters: SPN,COL,ROW
Sprite no. SPN is PUT onto the screen with its top left corner at COL,ROW

GET parameters: SPN,COL,ROW
GETs the sprite no. SPN from the screen at position COL,ROW

CPY parameters: SPN,SPN2
COPYs sprite no. SPN into sprite no. SPN2

The combinations of the four prefixes and four suffixes given yield the first sixteen commands given above.

The logical operations OR, AND and XOR will be familiar to anyone with a knowledge of Boolean algebra; however, it is easy to understand what they do in terms of pixels:

OR The destination pixel is set if either the source OR the destination pixel is set.

AND The destination pixel is set only if the source AND the destination pixels are set.

XOR The destination pixel is set only if one, but not both, of the source and destination pixels are set.

XOR is particularly useful when it is necessary to move a sprite over some background data - the sprite can be put on the screen using a PUTXOR and the background can be restored again with another PUTXOR.

Type in the following:

```
: TWOCOL MONO S2COL ATTON CLR ;
: INIT1 RED BLACK 0 SETATR 1 SPN ! 16 WID ! 16 HGT ! ISPRITE ;
: INITW .GREEN BLACK 0 SETATR 0 SPN ! SCLR 16 WINDOW ;
: POLY1 1 SPN ! 64 ROW ! 64 COL ! 64 WID ! 64 HGT ! 5 NUM ! 0 INC ! POLY ;
: POLY0 0 SPN ! 36 INC ! POLY ;
: PSET TWOCOL INIT1 INITW POLY1 POLY0 1 SPN ! 0 COL ! 0 ROW ! ;
```

TWOCOL puts the graphics into two-colour mode. INIT1 dimensions sprite no. 1 to be 16 character blocks square with black foreground and red background. INITW sets up the screen window. POLY1 draws a pentagon inside sprite no. 1, POLY0 draws one at an angle on the screen. Note that PSET sets SPN, COL and ROW, ready for a MOV command.

Type PSET<CR> then PUTBLK

You will see that sprite no. 1 is put on the screen, and it completely overwrites what was there before. Also, the attributes from sprite 1 are moved (i.e. black lines with red background) because of the ATTON in TWOCOL.

Type PSET<CR> again, and then PUTOR. Sprite 1 is "put on top of" any data that it is already on the screen.

After executing PSET again, type PUTAND. The only points that are left on the screen are where the lines from the two pentagons cross.

Type PSET again, followed by PUTXOR. The resulting display is similar to that from PUTOR, but the points are cleared where the lines cross. Type PUTXOR again, and the second pentagon vanishes, leaving the first one as it was before.

If you type ATTOFF after PSET, the screen colour will not be changed when you type the MOV command.

Type PSET ATTOFF PUTOR GETBLK. Sprite 1 now contains the double pentagon that is on the screen, and this can be placed on the screen at any point using PUTBLK. If you try to put the sprite wholly or partially off the screen (-1 COL ! -1 ROW ! PUTBLK), White Lightning does not give an error message, but places as much of the sprite on the screen as is possible. This automatic adjustment applies to all the data movement commands.

If you type 2 SPN ! 16 WID ! 16 HGT ! ISPRITE 1 SPN ! 2 SPN2 ! CPYBLK and PUT sprite 2 onto the screen, you will see that sprite 1 has been copied into sprite 2.

The MOV commands are much more general than the PUTs, GETs and CPYs and can be used to copy a window from any position in one sprite to any position in another. For example:

0 SPN ! 0 COL ! 0 ROW ! 4 WID ! 4 HGT ! 0 SPN2 ! 36 COL2 ! 12 ROW2 ! MOVBLK

will copy a 4x4 block from the top left-hand corner of the screen to halfway down the right-hand side.

When using AND,OR and XOR with multi-colour mode, the situation is more complex, and this is summarised below for the advanced user:

source colour	dest colour	dest. colour after BLK	OR	AND	XOR
0	0	**0**	0	0	0
0	1	**0**	1	0	1
0	2	**0**	2	0	2
0	3	**0**	3	0	3
1	0	**1**	1	0	1
1	1	**1**	1	1	0
1	2	**1**	3	0	3
1	3	**1**	3	1	2
2	0	**2**	2	0	2
2	1	**2**	3	0	3
2	2	**2**	2	2	0
2	3	**2**	3	2	1
3	0	**3**	3	0	3
3	1	**3**	3	1	2
3	2	**3**	3	2	1
3	3	**3**	3	3	0

(NB colour 0 means the background.)

TWO-WAY DATA MOVEMENT

The second set of sixteen data movement words are the two-way moves:

BLK%BLK	OR%BLK	XOR%BLK	AND%BLK
BLK%OR	OR%OR	XOR%OR	AND%OR
BLK%XOR	OR%XOR	XOR%XOR	AND%XOR
BLK%AND	OR%AND	XOR%AND	AND%AND

As you can see, they each consist of two of the logical operators BLK, OR, XOR or AND separated by a '%'.

The first operator specifies the logical operation for the data from SPN2 going into SPN, the second is the logical operation for the data going into SPN2 from SPN. All these commands have the same parameters:

SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

SPN,COL and ROW specify the top left-hand corner of window no. 1, while SPN2, COL2 and ROW2 give the top left-hand corner of window no. 2.

Data is moved between the two windows simultaneously, the logical operations operating in exactly the same way as with the single-way move commands.

MOVING ATTRIBUTES

Two words are provided to move blocks of attributes:

MOVATT parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
SWAPATT parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

They operate in a similar way to MOVBLK and BLK%BLK, moving the attributes only. If in ATTON mode, the primary set of attributes only is moved, while both primary and secondary sets are moved in ATT2ON mode.

COLLISION DETECTION

Collision detection is enabled by DTCTON and disabled by DTCTOFF. (It is always best to switch it off when not required since it slows down data movement slightly.)

The variables CCOL and CROW contain the column and row of the collision after sprite data has been moved. If a collision has not occurred, CCOL and CROW are both set to -1.

In the case of single-way move commands, CCOL and CROW will record the position in the destination sprite. With two-way moves CCOL and CROW give the position in SPN2. (Note that the collision is detected by examining the data before it is moved.)

The position in CCOL and CROW is the first collision found - the data is moved starting at the top left-hand corner going from left to right along each line of character blocks.

In multi-colour mode, colour1 and the background are both regarded as transparent to collisions - only colour2 and colour3 will result in a collision being detected.

CLEARING AND INVERTING WINDOWS

SCLR was mentioned earlier and is used to clear a whole sprite, setting the attributes to ATR. The secondary attributes are only set if in ATT2ON mode.

WCLR (parameters: SPN,COL,ROW,WID,HGT,ATR) is similiar to SCLR, but it only clears a window inside SPN rather than the whole sprite.

For example, if you put some data on the screen;

```
0 SPN ! 64 COL ! 64 ROW ! 64 WID ! 64 HGT ! 5 NUM ! 0 INC !
POLY 36 INC ! POLY
```

then the bottom right-hand corner of the figure is removed by:

```
8 ROW ! 8 COL ! 8 WID ! 8 HGT ! WCLR
```

SETA is used to initialise only the attributes in a window. As with SCLR and WCLR, the secondary attributes are not altered unless in ATT2ON mode.

For example try:

```
PURPLE BLACK 0 SETATR 0 COL ! 0 ROW ! 16 WID ! 16 HGT ! SETA
```

INV (parameters: SPN,COL,ROW,WID,HGT) will invert the pixel data in the specified window. If in MONO mode, background is changed to foreground and vice-versa. In MULTI mode, background is changed to colour 3, colour 1 is changed to colour 2 and vice-versa.

SCAN (parameters: SPN,COL,ROW,WID,HGT) will leave -1 (true) on the stack if there is data in the specified window. For example, 0 COL ! 0 ROW ! 16 WID ! 16 HGT ! SCAN . will give -1 if the screen is still set up with the display from the WCLR example.

ATTGET (parameters: SPN,COL,ROW) will look at the attribute value at the specified character block and place the attribute in the variable ATR.

SCROLLING COMMANDS

WHITE LIGHTNING has the following words for scrolling data and attributes in sprite windows:

```
SCR1    WRR1    SCL1    WRL1
SCR2    WRR2    SCL2    WRL2
SCR8    WRR8    SCL8    WRL8

SCROLL  WRAP

ATTUP   ATTDN   ATTL    ATTR
```

Type in the following words:

```
: SSET MONO S2COL ATTON 0 SPN ! 3 ATR ! SCLR 16 WINDOW ;
: SPOLY 32 ROW ! 32 COL ! 32 WID ! 32 HGT ! 4 NUM ! 0 INC ! POLY 0 NUM ! POLY ;
: STST SSET SPOLY 0 ROW ! 0 COL ! 8 WID ! 8 HGT ! ;
```

Now type STST <CR>; this puts some information on the screen to be scrolled. Note that STST also sets up ROW, COL, WID and HGT for scrolling.

The first twelve words scroll or "wrap" data left or right by 1,2 or 8 pixels. With a scroll, any data shifted off the edge is lost and blanks are shifted into the other side. In the case of a wrap, data which is shifted off one edge re-appears at the other side.

"SC" at the start of a word means "scroll" and "WR" means "wrap". The third letter is either "R" for right or "L" for left. The digit at the end is the number of pixels being scrolled in 2-colour mode. All 12 commands have the same parameters, SPN,COL,ROW,WID,HGT which define the window in which the scrolling is to take place. COL,ROW is the top left hand corner. None of these commands alter the attribute data.

If you type SCR1 you will see that the figure on the screen is shifted right by one pixel. If you now type : TEST1 63 0 DO SCR1 LOOP ; TEST1 <CR> the figure will be scrolled out of the window. Type STST followed by : TEST2 64 0 DO WRR1 LOOP ; TEST2 <CR>. The pixel data re-appears at the left of the window as it is shifted out of the right side.

The other ten commands in this group function in a similar way, only differing in the direction of scrolling or number of pixels scrolled. One WRR2 is not much faster than two WRR1s - the two-pixel scrolls are intended for use in multi-colour mode where one-pixel scrolls cannot be used.

WRAP and SCROLL are used to scroll vertically. The parameters are:

SPN,COL,ROW,WID,HGT,NUM

NUM is the number of pixels to be scrolled - a positive value indicates scrolling up and a negative value is used for scrolling down. The maximum value allowed is 127 (or -127).

ATTUP,ATTDN,ATTL and ATTR are used to scroll attributes by one character block up, down, left and right. The parameters for these commands are:

SPN,COL,ROW,WID,HGT

Type in the following:

```
: ATTSET1 RED BLACK 0 SETATR 0 COL ! 0 ROW ! 4 WID ! 4 HGT ! SETA ;
: ATTSET2 PURPLE BLACK 0 SETATR 4 COL ! 0 ROW ! 4 WID ! 4 HGT ! SETA ;
: ATTSET3 GREEN BLACK 0 SETATR 0 COL ! 4 ROW ! 4 WID ! 4 HGT ! SETA ;
: ATST SSET ATTSET1 ATTSET2 ATTSET3 0 COL ! 0 ROW ! 8 WID ! 8 HGT ! ;
```

Note that you need to have typed in the previous definition for SSET to use this example.

Typing ATST sets up some attributes to be scrolled.

You can see that the attributes are scrolled right one character block by ATTR. Note that the attributes are always wrapped round. ATTL (left), ATTUP (up) and ATTDN (down) are similar.

SPRITE TRANSFORMATIONS

This set of words are used to carry out transformations on sprites and are intended to be used prior to displaying sprites on the screen since they are not as fast as the move or scroll commands.

FLIP and FLIPA are provided to reflect a window around a horizontal line through its centre, and both have the following parameters:

```
SPN,COL,ROW,WID,HGT
```

FLIPA is used the reflect the attributes only; the secondary attributes are moved only if in ATT2ON mode. FLIP will move the pixel data, and attributes as well if the computer is in ATTON or ATT2ON mode.

Put some shapes on the screen:

```
0 SPN ! 16 COL ! 16 ROW ! 16 HGT ! 16 WID ! 0 NUM ! 0 INC ! POLY
48 COL ! 16 ROW ! 3 NUM ! POLY
16 COL ! 48 ROW ! 4 NUM ! POLY
48 COL ! 48 ROW ! 5 NUM ! POLY
```

If you now type:

```
0 COL ! 0 ROW ! 8 WID ! 8 HGT ! FLIP
```

you will see that the window is turned upside-down.

MIR and MAR are equivalent to FLIP and FLIPA; they have the same parameters but they reflect the window around a vertical line rather than a horizontal one.

SPIN is used to rotate one window through 90 degrees into another window - unlike FLIP and MIR, it cannot be used with multi-colour mode data. Its parameters are:

```
SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
```

WID and HGT are the width and height of the source window; obviously, these are interchanged to give the dimensions of the destination window in SPN2.

If you now type:

```
0 SPN2 ! 8 COL2 ! 8 ROW2 ! SPIN
```

the data you put on the screen earlier will be rotated and placed on a different part of the screen.

XPANDX and XPANDY expand one sprite window into another in the X (horizontal) and Y (vertical) directions respectively. They both use the following parameters:

```
SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
```

WID and HGT give the size of the source window; in the case of XPANDX, the destination window will be twice as wide as WID, and with XPANDY, the destination will be twice as high as HGT.

Since expansion of the window begins at the right in the case of XPANDX and at the bottom of the window for XPANDY, it is possible to expand a window into itself. Try:

```
0 COL2 ! 0 ROW2 ! XPANDX
```

followed by

```
16 WID ! XPANDY
```

CHARACTER MANIPULATION

LCASE and UCASE put the text into lower case and upper case respectively. Since WHITE LIGHTNING holds the character set in RAM, the new character set has to be copied down from the character ROM.

CHAR moves a character into a sprite at a specified row and column. The parameters are:

```
SPN,COL,ROW,NUM
```

NUM is the number of the character. As you may already know, the C64 uses display codes when displaying data on the screen which differ from the ASCII codes. The display codes refer directly to the position in the character set. NUM is the number of the character being used:

0 to 255	ordinary ASCII characters
256 to 511	reverse ASCII characters
512 to 767	display codes
1024 to 1279	double width ASCII
1280 to 1535	reverse double width ASCII
1536 to 1791	double width display codes

If NUM is less than 256, it is assumed to be an ASCII character, and is converted into display codes 0 to 127 which are normally the non-reversed characters. If NUM is in the range 256 to 511, the display codes from 128 to 255 are used instead, which are normally reverse characters. When NUM is between 512 and 767, 512 is subtracted to give the display code.

It is also possible to put double-width characters into a sprite, using NUM>1024.

For example, an "A" can be placed on the screen using either 0 SPN ! 0 COL ! 0 ROW ! 65 NUM ! CHAR (65 is the ASCII code for "A") or 0 SPN ! 0 COL ! 0 ROW ! 513 NUM ! CHAR (513 - 512 = 1 is the display code for "A").

V" is similar to ." and can be used to place a whole string on the screen; the parameters are:

```
SPN,COL,ROW
```

The word +V" can be used to set an offset which is added to the ASCII value of each character in the string before placing it in the sprite. Thus an offset of zero gives ordinary characters, 256 gives reverse characters, 1024 gives double width and 1280 gives reverse double-width. Also, an offset of 2048 will put single-width characters in the sprite, but will leave a character block between each one; 2304 will give double-spaced reverse characters. Try the following:

```
0 SPN ! 10 COL ! 10 ROW !
0 +V" V" WHITE LIGHTNING"
256 +V" V" WHITE LIGHTNING"
1024 +V" V" WHITE LIGHTNING"
1280 +V" V" WHITE LIGHTNING"
```

PUTCHR is the opposite of CHAR - it moves a character block from a sprite back into the character memory. It has the same parameters as CHAR:

SPN,COL,ROW,NUM

NUM can take on a value between 0 and 767, 0 to 255 converting NUM from ASCII, 256 to 511 giving reverse ASCII, and 512 to 767 being converted straight into the display code, as with CHAR. Note that if you want to use PUTCHR to redefine the letter "A" for example, you must also redefine the reverse "A" (display code 129) if it is to be used with the flashing cursor.

Finally, DBLANK will blank the entire display to border colour while DSHOW will turn on the display again.

READING THE KEYBOARD, JOYSTICK AND LIGHTPEN

KEYBOARD

It is of course possible to read the keyboard using KEY, but as it always waits for a key to be pressed, this cannot detect multiple key depressions and cannot detect the shift or Commodore keys.

n KB will leave a true value (-1) on the stack if key number n is pressed. The keys are numbered from 0 to 63:

0	INST/DEL	32	f1
1	"3"	33	"Z"
2	"5"	34	"C"
3	"7"	35	"B"
4	"9"	36	"M"
5	"+"	37	"."
6	" "	38	R.H.SHIFT
7	"1"	39	SPACE BAR
8	RETURN	40	F3
9	"W"	41	"S"
10	"R"	42	"F"
11	"Y"	43	"H"
12	"I"	44	"K"
13	"P"	45	":"
14	"*"	46	"="
15	" " top l.h. of K.B.	47	'COMMODORE' KEY
16		48	f5
17	"A"	49	"E"
18	"D"	50	"T"
19	"G"	51	"U"
20	"J"	52	"O"
21	"L"	53	"@"
22	";"	54	" "
23	CTRL	55	"Q"
24	f7	56	
25	"4"	57	SHIFT LOCK and L.H. shift
26	"6"	58	"X"
27	"8"	59	"V"
28	"0"	60	"N"
29	"-"	61	","
30	CLR/HOME	62	"/"
31	"2"	63	RUN/STOP

JOYSTICKS

FIRE1 and FIRE2 are functions which will give a true value on the stack if the fire button on the joystick in control ports 1 or 2 is pressed. For example:

```
.
.
.
FIRE1 IF ZAPALIEN THEN
.
.
.
```

JS1 and JS2 give the directions of joysticks 1 and 2 respectively; the direction is represented as a number from 0 to 8:

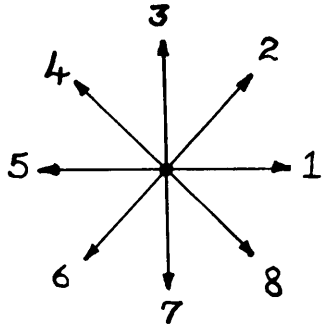


Fig 3

LIGHTPEN

LPX and LPY give the X and Y positions of the lightpen respectively.

HARDWARE SPRITES

Besides having its own software sprites, WHITE LIGHTNING allows the use of the Commodore's hardware sprites.

Unlike software sprites, hardware sprites are not controlled using PUTs and GETs; each sprite can appear at only one location on the screen and is separate from the pixel data. In addition, each hardware sprite carries its own colour (independent of the attributes) and display mode (2 colour or 4 colour), and can be displayed with either hi-res or text data.

Defining a Hardware Sprite

Each hardware sprite is 24 pixels long and 21 pixels high, occupying 63 bytes in memory. Due to memory constraints, the sprite definitions must share memory with the character set, each sprite using the same amount of space as eight characters (one byte is left unused at the end of each definition to make 64 bytes). Since the character set takes up 2k bytes (from $C000 to $C7FF), there is room for

2048/64 = 32 definitions, which are numbered from 0 to 31, although only 8 can be put on the screen at once. Sprite definition number n will share memory with the characters whose display codes are n*8 to n*8+7.

The data for a hardware sprite is designed with the Sprite Generator Program and saved as a software sprite in the normal way. At run-time, the pixel data can be copied from any part of a software sprite into a hardware sprite using SPRCONV.

SPRCONV has the following parameters:

```
SPN,COL,ROW,SPN2
```

SPN, COL and ROW define the top left-hand corner of the source software sprite in the normal way, COL and ROW being in pixels. SPN2 is the hardware sprite definition number. As an example type in the following:

```
256 +V"
0 SPN ! 0 COL ! 0 ROW !
V" *QA" 1 ROW ! V" SIS"
2 ROW ! V"    "
0 ROW ! 16 SPN2 ! SPRCONV
```

Since the information was put into hardware sprite definition no. 16, the eight characters with display codes from 8*16 = 128 onwards will have been overwritten. To verify this, type CTRL-9 to give reverse characters, followed by "ABCDEFG".

Only eight hardware sprites can exist on the screen at one time, each of which can be associated with one of the 32 definitions using .SET :

```
definition# sprite# .SET
```

The sprites are numbered from 0 to 7. If you now type "16 1 .SET", hardware sprite no. 1 will be associated with the definition which you have just created.

It is of course possible to copy a large software sprite into several hardware sprites.

Switching on a Hardware Sprite

Before a sprite can be displayed, it must be turned on using .ON:

```
sprite# .ON
```

1 .ON will enable sprite no. 1. The equivalent word to turn a sprite off again is:

```
sprite# .OFF
```

To define a sprite's colour, use:

```
colour sprite# .COL
```

BLACK 1 .COL will make sprite no. 1 black.

Placing a Sprite on the Screen

Once the sprite has been enabled and given a colour, it will still not be visible because it is positioned off the screen. Positioning of a sprite on the screen is carried out by 'position sprite# .XPOS' and 'position sprite# .YPOS', the positions in both cases being at pixel resolution. Sprite number 1 can be placed at the top left of the screen using:

```
24 1 .XPOS
50 1 .YPOS
```

Values of the x co-ordinate between 1 and 23 allow the sprite to be positioned partially off the screen; similarly for y co-ordinates between 30 and 49.

Moving a hardware sprite around the screen is very easy:

```
: TEST3 250 0 DO I 1 .XPOS LOOP ;
100 1 .YPOS TEST3
```

Double-Sized Sprites

It is possible to expand a hardware sprite to double size in either direction using sprite# .XPANDX to expand in the X-direction and sprite# .XPANDY to expand in the Y-direction. If you type 1 .XPANDX followed by 1 .XPANDY, the sprite on the screen will be expanded to double size.

sprite# .SHRINKX and sprite# .SHRINKY have the opposite effect, returning a sprite to normal size.

A double-sized sprite is partially displayed on the screen with y-co-ordinates between 9 and 14, or x-co-ordinates from 481 to 503, then 0 to 24.

Multi-Coloured Sprites

A hardware sprite is put into multi-colour mode using sprite# .4COL. As with the hi-res screen, horizontal resolution is cut in half. Two more colours are required; these are the same for all sprites and are set by .COL0 and .COL1:

```
colour .COL0
colour .COL1
```

The four possible colours are displayed differently by a hardware sprite than by the hi-res screen:

Hi-Res Screen (Software Sprite)	Hardware Sprite
Background colour	transparent (screen colour)
Colour 1	colour zero (set by .COL0)
Colour 2	sprite colour (set by n .COL)
Colour 3	colour 1 (set by .COL1)

A sprite can be put back into 2 colour mode using sprite# .2COL

Display Priorities

Lower numbered sprites have priority over high numbered sprites, e.g. sprite 0 will always appear to pass in front of sprite 1 if they coincide. It is also possible to control the priorities between sprites and background data (i.e. the hi-res pixel data created by the software sprites). To give the background priority over a sprite, use 'sprite# .OVER', whilst 'sprite# .UNDER' puts the background underneath a sprite again, as normal.

Hardware Sprites Collision Detection

Collisions between two sprites or between a sprite and background data is detected using n .HIT. If n is less than 8, it leaves (-1) on the stack if sprite n has hit another sprite. If n is greater than 8, -1 is left if sprite n-8 has hit background data. When you use .HIT with a value of less than 16, the records of any other sprite-to-sprite or background-to-sprite collisions are cleared. However, you can still detect these by adding 16 to n, in which case the value of the sprite collision register the last time that n was less than 16 is used.

In multi-colour mode, colour zero (set by .COL0) and background colour 1 are considered to be transparent for collisions.

SMOOTH SCROLLING

Smooth scrolling allows you to shift the entire screen over by 1 to 7 pixels horizontally or vertically. Once the screen has been shifted over by seven pixels, a wrap or scroll command must be used to move it by one character.

Before using smooth scrolling, the screen must be shrunk to 38 columns by 24 rows to give space for new data to be shifted in; this is achieved using the word H38COL (to go back into normal display mode, use H40COL). This gives two columns on either side of the screen which are hidden under the border, and one at the bottom of the screen.

Using n SCRLX and n SCRLY, where n is between 0 and 7. it is possible to shift the entire screen by n pixels.

4. SOUND

WHITE LIGHTNING provides a set of sound commands which allow you to control the 64's SID chip.

To generate a sound from one of the three voices in the SID chip, you need to set up the following:

1. Master volume
2. Frequency
3. Envelope (ADSR)
4. Waveform

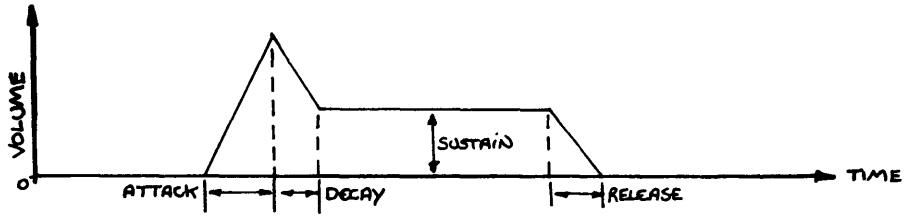
The master volume of the SID chip is set using VOLUME - it takes a value off the stack which should be in the range 0 to 15. For example, "15 VOLUME" sets the volume to its maximum value.

The frequency of a voice is set by FRQ which takes the following form:

```
frequency voice FRQ
```

The voice is either 1, 2 or 3. The frequency is not in Hz (cycles per second); you must multiply the frequency in Hz by 16.4015 first. This can be done using 4084 249 */ Thus, to set the frequency of voice 1 to A (440 Hz) you could use 440 4084 249 */ 1 FRQ. The maximum value of the frequency is 65535.

The volume of a musical note changes from when it is first struck. This can be split into four phases: attack, decay, sustain and release:



After being struck, the volume rises to its peak value at a rate determined by the 'attack'. It then falls to the 'sustain' level at a rate determined by the 'decay'. At the end of the note, the volume falls away to zero at the 'release' rate. This 'envelope' shape can be set up using ADSR: attack decay sustain release voice ADSR. The voice is 1, 2 or 3, and the rest of the parameters are in the range 0 to 15.

The time between the start of the attack and the start of the release is set by MUSIC which has two parameters; the voice number and the length of the note. The length can be from 1 to 255 and is measured in 60ths of a second. A value of 0 indicates that the note lasts indefinitely:

```
length voice MUSIC
```

As an example, type in the following:

```
SIDCLR 15 VOLUME 7217 1 FRQ
5 8 5 9 1 ADSR 1 TRI 20 1 MUSIC
```

SIDCLR simply clears any values that were set up in the sound chip. TRI sets up the waveform type and is explained in the next section.

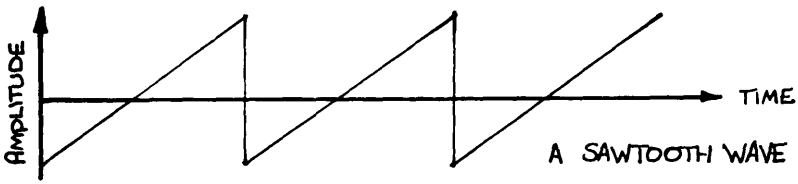
Try experimenting with different values of frequency and other parameters for ADSR.

Changing the Waveform

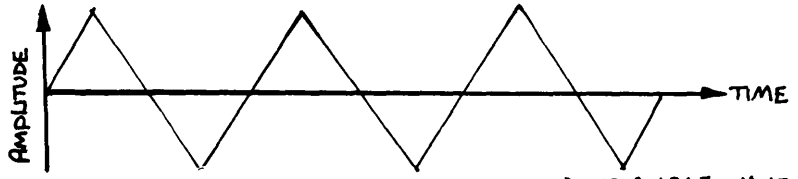
The 64's sound generator is capable of generating four types of waveform:

1. SAWTOOTH waves
2. TRIANGLE waves
3. PULSE waves
4. NOISE

'SAWTOOTH' and 'TRIANGLE' refer to the shape of the waveform when plotted on a graph:

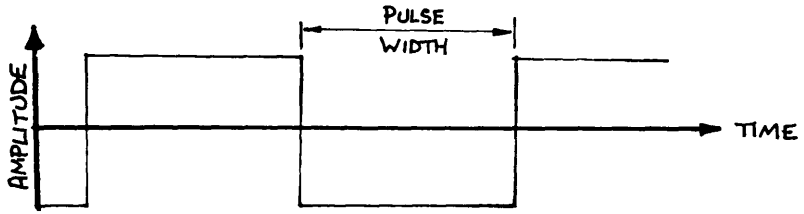


A SAWTOOTH WAVE



A TRIANGLE WAVE

A 'PULSE' wave looks like this:



The pulse width can be varied, so that a variety of sounds can be created.

NOISE can be used to generate realistic explosions.

SAW selects the sawtooth waveform:

```
voice SAW
```

TRI selects triangle waves:

```
voice TRI
```

NOISE selects noise:

```
voice NOISE
```

PULSE selects a pulse wave:

```
width voice PULSE
```

The pulse width is in the range 0 to 4095; 2048 gives a square wave.

Try changing the last example to use different waveforms, and experiment with different values of pulse width when using the PULSE waveform.

Here are some examples of the sounds that can be created:

GUNFIRE:

```
15 VOLUME 7217 1 FRQ 1 9 3 9 1 ADSR 1 NOISE 20 1 MUSIC
```

EXPLOSION:

```
15 VOLUME 3000 1 FRQ 0 13 5 12 1 ADSR 1 NOISE 20 1 MUSIC
```

DEPARTING UFO:

```
SCR #1
 0 ( DEPARTING UFO ...)
 1 : UFO SIDCLR
 2 15 VOLUME
 3 9 2 11 12 1 ADSR
 4 1 TRI
 5 30 1 MUSIC
 6 31 1 DO
 7   28000 12000 DO
 8      I 1 FRQ
 9   320 +LOOP
10 LOOP ;
11 ;S
12
13
14
15
```

Any of these examples could, of course, use voice 2 or 3.

FILTERING

The timbre of sound produced can be altered using filtering. Using FILTER it is possible to control whether the output from each oscillator is passed through the filter or not. The format for this is:

```
flag voice FILTER
```

The flag is either 1 or 0, 1 indicating that the voice is to be filtered and 0 indicating that it is not.

n PASS selects the filter's mode of operation:

```
0 PASS low pass
1 PASS high pass
2 PASS band pass
3 PASS notch reject
```

In low pass mode, frequencies above the cut-off frequency are attenuated. In high pass mode, frequencies below the cut-off frequency are attenuated. In band pass mode, only a narrow band around the cut-off is passed while notch reject has the opposite effect.

n CUTOFF is used to select the cut-off frequency. The frequency is in the range 0 to 2047; i.e. the frequency used by FRQ must first be divided by 32 before being used with CUTOFF.

It is also possible to make the filter resonant around the cut-off frequency using n RESONANCE; the parameter is in the range 0 to 15.

Using a low pass filter in conjunction with resonance, the 'EXPLOSION' example given earlier can be made more realistic:

```
15 VOLUME 3000 1 FRQ 0 13 5 12 1 ADSR
1 NOISE 1 1 FILTER 0 PASS
120 CUTOFF 12 RESONANCE 20 1 MUSIC
```

RING MODULATION AND SYNCHRONISATION

Using ring modulation of two voices, very complex waveforms can be produced. Ring modulation is enabled using RING:

```
flag voice RING
```

The flag is 1 or 0, enabling or disabling ring modulation respectively.

If voice=1, voice 1's output is replaced by voice 1 ring modulated with voice 3. When voice=2, voice 2 is ring modulated with voice 1. Voice 3 is ring modulated with voice 2 when voice=3.

Realistic bell effects can be generated by using ring modulation coupled with low pass filtering:

```
SCR # 1
  0 ( BELL EXAMPLE ...)
  1
  2 0 VARIABLE BPTR
  3
  4 CREATE BDATA
  5 1514 , 10500 , 1201 , 10500 , 1340 , 10500 , 900 , 21000 ,
  6 900 , 10500 , 1348 , 10500 , 1514 , 10500 , 1201 , 21000 ,
  7 1201 , 22500 , 1201 , 22500 , 1201 , 22500 , 1201 , 22500 ,
  8 1201 , 22500 , 1201 , 22500 , 1201 , 22500 , 1201 , 22500 ,
  9 1201 , 22500 , 1201 , 22500 , 1201 , 22500 , 1201 , 22500 ,
 10 -1 , SMUDGE
 11 -->
 12
 13
 14
 15

SCR # 2
  0 : BSET SIDCLR 15 VOLUME 1 9 8 12 1 ADSR
  1 3 TRI 1 TRI 1 1 RING 1 1 FILTER 0 PASS 80 CUTOFF ;
  2 : STRIKE DUP 1 FRQ 506 256 */ 3 FRQ 40 1 MUSIC ;
  3
  4 : BPLAY BEGIN
  5     BPTR @ @ DUP 1+
  6   WHILE
  7     2 BPTR +! STRIKE
  8     BPTR @ @ 0 DO LOOP
```

```
 9     2 BPTR +!
10   REPEAT DROP ;
11
12 : BELL BSET ' BDATA BPTR ! BPLAY ;
13 ;S
14
15
```

After you have typed in and LOADed the above screens type BELL <CR>.

Synchronisation of two voices is enabled using SYNC in a similar way to RING: flag voice SYNC.

Synchronisation of two voices can be used to mimic the sound of engines. In the following example we hear what happens when a hedgehog is run over ...

```
SCR # 1
  0  ( BIKE SQUEAL)
  1
  2  0 VARIABLE EPTR
  3  CREATE BIKEDATA
  4  15 , 750 , 270 , 15 , 1000 , 650 ,
  5  15 , 1150 , 750 , 21 , 1400 , 1000 , -1 , SMUDGE
  6  : READ EPTR @ @ 2 EPTR +! ;
  7  : BIKE SIDCLR 15 VOLUME 15 5 15 15 1 ADSR
  8  1 TRI 3 TRI 1 1 SYNC ' BIKEDATA EPTR !
  9  BEGIN READ DUP 1+ WHILE 100 1 MUSIC
 10  READ READ DO I 3 FRQ I 5 2 */ 1 FRQ
 11  DUP 0 DO LOOP LOOP DROP REPEAT DROP ;
 12  : SQUEAL 0 1 SYNC 1 1 RING 3 5 8 3 1 ADSR
 13  60 1 MUSIC 5000 0 DO I 20000 + 1 FRQ
 14  I 20250 + 3 FRQ 35 +LOOP ;
 15  -->

SCR # 2
  0  ( CRASH, SIREN, EX4)
  1
  2  : CRASH 0 1 RING 3000 2 FRQ 0 13 5 12 2 ADSR
  3  2 NOISE 1 2 FILTER 0 PASS 120 CUTOFF
  4  15 RESONANCE 20 2 MUSIC 5000 0 DO LOOP ;
  5  : SIREN 10000 8409 1000 1 PULSE 15 15 15 15 1 ADSR
  6  255 1 MUSIC 32 0 DO OVER 1 FRQ
  7  3500 0 DO LOOP DUP 1 FRQ 3500 0 DO LOOP
  8  I 7 = IF DROP DROP 9803 8244 THEN
  9  LOOP DROP DROP ;
 10  : EX4 BIKE SQUEAL CRASH SIREN ;
 11
 12
 13
 14
 15  ;S
```

After you have typed in and LOADed the above screens, type EX4 <CR>.

MUTE, OSC and ENV

OSC and ENV leave on the stack the output amplitude of voice 3's oscillator and envelope generator. It is possible to obtain a vibrato effect by using OSC to modify either voice 1 or voice 2's frequency.

If oscillator 3 is being used in this way its output must be disabled using MUTE. 1 MUTE disables oscillator 3's output and 0 MUTE enables it again.

SPRITE STORAGE ORGANISATION

After WHITE LIGHTNING has loaded, 6K of space is automatically allocated for sprite storage. When a sprite is created using ISPRITE the start of sprites pointer is moved down, and when deleting a sprite it is moved up, thus the end of sprite space pointer, SPND, is normally pointing to the same location all the time. CLR sets SPST to SPND-1 and then stores a dummy zero, thus deleting all the sprites.

In the tape version of White Lightning, the space below sprites is used to hold source code.

If you need more than 6k for sprites then use the LOMEM word as described in the operating instructions. LOMEM takes an address from the stack and uses it as the lowest permissible address for sprites. It must be on a page boundary.

Screen 0 is used by the editor and occupies $6400 to $6800. The minimum value for LOMEM is $6C00 which allocates one screen for source. The maximum configuration for source (minimum configuration for sprites) has LOMEM set to $9800 which actually leaves only 256 bytes for sprites.

If for some reason you want to build sprites upwards in memory instead of downwards, you can use the words SPRITE, WIPE and RESET which are analogous to ISPRITE, DSPRITE and CLR.

To reserve memory from 8000 upwards for sprites, use the following:

```
HEX 8000 LOMEM 8000 SPST ! RESET
```

When SPRITE, WIPE and RESET are used, SPST will remain the same, and SPND will change (up in the case of SPRITE, and down in the case of WIPE and RESET.)

To move all the sprites up or down in memory, you can use RELOCATE which moves all sprites by an offset held in NUM. SPST and SPND are altered, but the top and bottom of reserved memory are not. If you try to RELOCATE out of the reserved memory, a "? NO ROOM ERROR" is reported.

For example,

```
HEX 400 NUM ! RELOCATE
```

will move all sprites up in memory by 1K.

```
HEX -400 NUM ! RELOCATE
```

will move all sprites down by 1K.

When you load in sprites from tape or disk using RECALL or MERGE, the sprites are loaded in at the bottom of the reserved memory then relocated upwards so that all the sprites reside at the top of memory.

VARIABLE SETS

White Lightning actually uses eight sets of the sprite variables SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2,INC,ATR,CCOL and CROW. These are accessed using n SET, where n is 0 to 7. Normally White Lightning uses set 0. To see how variable sets work, try the following:

```
0 SET 1 SPN ! SPN ?
```

The number 1 should be printed, as this is the contents of SPN. Now type:

```
1 SET 3 SPN ! SPN ?
```

This selects variable set no. 1, rather than 0 which was used before, and stores 3 in SPN.

If you now type:

```
0 SET SPN ?
```

you will see that "1" is printed, i.e. the two variable sets are distinct. It is possible to speed up time-critical parts of a program by setting up parameters for different graphics words in the various variable sets beforehand and then switching very rapidly between them.

USING INTERRUPTS

One of White Lightning's most powerful features is the ability to execute a Forth word under interrupt. This is enabled using INT-ON:

```
' WORD INT-ON
```

where "WORD" is the word to be executed under interrupt.

When an interrupt occurs, every 60th of a second, 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 sixtieth of a second, it is not possible to execute it more than 30 times a second; if it exceeds a thirtieth of a second, it can only be executed 20 times a second, and so on. There is, however, no limit to the length of the background execution time itself. Secondly, as the execution time approaches a sixtieth of a second, or some multiple of it, then less and less processor time will be available for the foreground 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 be taken by disabling and re-enabling the interrupt using the words 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 half-way 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 the background re-enabled using EI ready for the next interrupt to occur:

```
DI PUTBLK EI
```

Variable set number 7 is used by the background program running under interrupt. Thus, to set a 4 by 4 character square scrolling at the top left hand corner of the screen, you would use:

```
0 SET 0 SPN ! 16 ROW ! 16 COL !
16 WID ! 16 HGT ! 4 NUM ! POLY
7 SET 0 SPN ! 0 ROW !
0 COL ! 4 WID ! 4 HGT ! ' WRR1 INT-ON
```

To stop execution of the background word, use INT-OFF.

Of course, any word can be executed under interrupt, including one that you have defined yourself. For example, two windows on the screen could be scrolled like this:

```
7 SET 1 NUM ! 0 SPN ! 0 ROW ! 0 COL ! 2 WID ! 4 HGT !
6 SET -1 NUM ! 0 SPN ! 0 ROW ! 2 COL ! 2 WID ! 4 HGT !
: FRED WRAP 6 SET WRAP ;
' FRED INT-ON
```

Note the use of variable sets to speed up execution.

Suppose we wanted to execute the word FRED less than 60 times a second. This can be done by counting interrupts. Try the following:

```
1 VARIABLE LCNT 0 VARIABLE ICNT
: JIM 1 ICNT +! ICNT @ LCNT @ > IF 1 ICNT ! FRED THEN ;
```

The word JIM increments ICNT, and if it is already greater than LCNT it sets ICNT back to 1 and executes FRED.

Now type ' JIM INT-ON. Because LCNT is set to 1, FRED will execute every sixtieth of a second, as before. If you now type 5 LCNT ! the word FRED only executes every twelfth of a second. Incrementing LCNT will slow down execution of FRED, but decrementing it will speed it up.

Interrupt-driven words: PLAY, RPLAY, TRACK and MOVE

These four words allow you to play tunes or move hardware sprites around the screen, under interrupt, the necessary data being taken from a software sprite.

PLAY and RPLAY

These are used to play tunes under interrupt and both have three parameters: SPN, COL and ROW. Unlike normal usage, COL and ROW are both sprite numbers. Sprite

SPN is the software sprite with data for voice 1, sprite COL contains voice 2's data and sprite ROW contains voice 3's data.

For example, if you wanted the data in sprite 3 to be played by voice 1, sprite 9 by voice 2 and sprite 15 by voice 3, you would use:

```
3 SPN ! 9 COL ! 15 ROW ! PLAY
```

It is also possible to keep a voice silent (so that it could be used to generate sound effects with MUSIC) by specifying sprite zero:

```
3 SPN ! 0 COL ! 15 ROW ! PLAY
```

- this would keep voice 2 silent.

Using this method it is possible to silence all the voices:

```
0 SPN ! 0 COL ! 0 ROW ! PLAY
```

If you use PLAY, the voices will remain silent after the last byte of data has been read from the sprite. This can be solved by using RPLAY, in which case, the tune is repeated until stopped as in the previous example.

Format for Storing Tunes in Sprites

Each note inside the sprite takes up four bytes - two bytes frequency in the usual low byte-high byte order followed by the length of the note in sixtieths of a second, and the time (again in sixtieths of a second) taken between releasing the present note and striking the next one.

The data is contained in the pixel data part of the sprite only - the attribute part is not used. Since each character block uses up 8 bytes, 2 notes will fit into one character block. Thus, if the tune contained 30 notes, a 15x1 sprite could be used to store the data.

There are two ways of putting the data into a sprite - it can either be put there using the sprite generator or it can be put there directly. The BASIC Lightning manual shows how to put data into a sprite from BASIC.

Before using PLAY or RPLAY, the waveform, volume and envelope must be set up as for MUSIC.

TRACK

This is similar to PLAY in that it reads data from a software sprite under interrupt - however, in this case the data is used to move a hardware sprite around the screen. The data is held in groups of two bytes inside the sprite; one byte x-offset followed by one byte y-offset. These offsets are signed (-128 to 127) and are added to the sprites x and y position on the screen every fiftieth of a second.

TRACK has two parameters: SPN and SPN2. SPN is the software sprite containing the data and SPN2 is the hardware sprite to be moved (0 to 7). As with PLAY, the data can be put in the sprite using the sprite generator, or it can be put there directly.

Since each character block contains data for 4 interrupts, and the interrupts occur 50 times a second, a 15x1 sprite would be required to hold 1.2 second's animation.

TRACK deals with offsets which are added to the current position on the screen, so it can be used to carry out animation from any starting position. Before using TRACK, the sprite must be turned on and positioned in the normal way.

MOVE

MOVE (parameters SPN,COL,ROW) is similar to TRACK - however, it only allows movement by a constant amount, and data is not read from a sprite. SPN is the hardware sprite, COL is the x-offset and ROW is the y-offset.

If a hardware sprite moves off the screen, it is automatically halted and turned off. Also, if you remove a sprite from the screen using .OFF, any animation that was taking place will stop.

It is possible to use n @XPOS and n @YPOS to read a sprite's position:

```
3 .HIT IF 3 @XPOS 3 @YPOS EXPLODE THEN
```

Using the Glossary

In describing operations on windows,

W1 XOR W2 -> W1

would mean:

"Window 1 exclusive-ored with window 2 goes into window 1"

'S1' and 'S2' are used in the same way, meaning 'sprite one' and 'sprite two'

There are essentially two types of IDEAL word; those which take their parameters from the stack, and those which use the IDEAL variables as parameters.

Where a word uses the stack the format is

parameter1 parameter2 ..... parametern WORD

for example

Colour Sprite# .COL

tells you that the word .COL has two parameters; the first is a colour and the second is a sprite number. It also tells you that the colour should be stacked first and that the sprite number should be stacked on top of that.

Where a word uses IDEAL variables, the parameter list is displayed to the right of the word in the form:

WORD Parameters: parameter1, parameter2, ...

for example

SCL1 Parameters: SPN,COL,ROW,WID,HGT

Tells you that the word SCL1 will use the values held in the IDEAL variables SPN,COL,ROW,WID, and HGT and that they must therefore be set up before the word is executed. The stack is unaffected by it's execution.

IDEAL variables:

SPN	COL	ROW	WID	HGT
SPN2	COL2	ROW2		
NUM	INC	ATR		
CCOL	CROW			
SPST	SPND			

THE WORDS

Sprite# .2COL

Puts a hardware sprite (number is zero to 7) into two-colour mode.

Sprite# .4COL

Puts a hardware sprite into 4-colour mode.

Colour Sprite# .COL

Sets the colour of a hardware sprite. (colour is 0..15).

Colour .COL0

Sets multicolour sprite colour #0.

Colour .COL1

Sets multicolour sprite colour #1.

Sprite# .HIT

Leaves a true value on the stack if the hardware sprite has hit something.

Sprite# .OFF

Removes a hardware sprite form the screen.

Sprite# .ON

Turns on a hardware sprite.

Sprite# .OVER

Background is given priority over the hardware sprite.

definition Sprite# .SET

Associates a hardware sprite with its definition.

Sprite# .SHRINKX

The enlarged sprite is given normal size in the X-direction.

Sprite# .SHRINKY

The enlarged sprite is given normal size in the Y-direction.

Sprite# .UNDER

The background goes under the hardware sprite.

Sprite# .XPANDX

The sprite is expanded to double size in the X-direction.

Sprite# .XPANDY

The sprite is expanded to double size in the Y-direction.

Position Sprite# .XPOS

Sets up the X position of hardware sprite.

attack decay sustain release voice ADSR

Specify envelope

Position Sprite# .YPOS

Sets up Y position of hardware sprite.

AFA Parameters: SPN

Leaves the attribute field address of sprite on the stack. -1 if non existant.

AFA2 Parameters: SPN

Leaves the 2ndary attribute field address of a sprite on the stack. -1 if non-existant.

AND%AND Parameters: SPN,ROW,COL,HGT,WID,SPN2,ROW2,COL2

```
W1 AND W2 ->W1
W1 AND W2 ->W2
```

AND%BLK Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

```
      W1->W2
W1 AND W2->W1
```

AND%OR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

```
 W1 OR W2 ->W2
W1 AND W2 ->W1
```

AND%XOR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

```
W1 XOR W2 ->W2
W1 AND W2 ->W1
```

ATT2ON

Enables movement of both sets of attributes with the data movement commands.

ATTDN Parameters: SPN,COL,ROW,WID,HGT

Scrolls down all attributes in a window by 1 character block with wrap.

ATTGET Parameters: SPN,COL,ROW

Puts the attribute at the specified position into ATR.

ATTL Paramters: SPN,COL,ROW,WID,HGT

Scrolls attributes in a window left by one character with wrap.

ATTOFF

Disables movement of attributes when pixel data is moved.

ATTON

Enables movement of primary set of attributes only.

ATTR Parameters: SPN,COL,ROW,WID,HGT

Scrolls attributes in window right with wrap.

ATTUP Parameters: SPN,COL,ROW,WID,HGT

Scrolls attributes in window up with wrap.

BLK%AND Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

W1 AND W2 ->W2
W2 -> W1

BLK%BLK Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

W1->W2
W2->W1

BLK%OR Parameters: SPN,ROW,COL,HGT,WID,SPN2,ROW2,COL2

W1 OR W2 ->W2
W2 ->W1

BLK%XOR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

W1 XOR W2 ->W2
W2 -> W1

BOX Parameters: SPN,COL,ROW,WID,HGT

Fills a block of pixels inside a sprite - this command is dependant on the current value of MODE.

CHAR Parameters: SPN,COL,ROW,NUM

Puts a character in a sprite at character block position specified.

Value of NUM

0 to 255	ASCII characters
256 to 511	reverse ASCII characters
512 to 767	display codes
1024 to 1279	double width ASCII characters
1280 to 1535	reverse double width ASCII characters
1536 to 1791	double width display codes

CLR

Erase all Sprites and set Sprite Storage to top of memory.

CPYAND Parameters: SPN,SPN2

S1 AND S2 -> S2

CPYBLK	SPN,SPN2	S1 -> S2
CPYOR	SPN,SPN2	S1 OR S2 -> S2
CPYXOR	SPN,SPN2	S1 XOR S2 -> S2

frequency CUTOFF

Set cutoff frequency (0..2047) for filter.

DBLANK

Blanks the screen to border colour.

DFA Parameters: SPN

Leaves pixel data address of a sprite on the stack. -1 if non-existant.

DRAW Parameters: SPN,COL,ROW,COL2,ROW2

Draws a line from (COL,ROW) to (COL2,ROW2).

DSHOW

Enable screen display (opposite of DBLANK)

DSPRITE Parameters: SPN

Delete Sprite, changing SPST

DTCTOFF

Turn off collision detection

DTCTON

Turn on collision detection

ENV

Leave output from oscillator 3 envelope generator on stack.

flag voice FILTER

Enable/disable filtering of a voice.

FIRE1

Leave true flag on stack if joystick in port 1 has fire button pressed.

FIRE2

Leave true flag on stack if joystick in port 2 has fire button pressed.

FLIP Parameters: SPN,COL,ROW,WID,HGT

Flip over window top to bottom.

FLIPA Parameters: SPN,COL,ROW,WID,HGT

Flip over attributes top to bottom.

frequency voice FRQ

Set frequency

GETAND Parameters: SPN,COL,ROW

Copy screen at (COL,ROW) into SPN with AND.

Similarly, GETBLK, GETOR and GETXOR

H38COL

Shrink display.

H40COL

Expand display to normal size

colour HBORDER

Sets border colour for hi-resolution screen.

HIRES

Go into hires mode.

COLOUR HPAPER

Set hi-res background colour (applies to 4-colour mode only).

Colour INK

Set INK colour for printing.

INV Parameters: SPN,COL,ROW,WID,HGT

Invert window.

ISPRITE Parameters: SPN,WID,HGT

Create Sprite, changing SPST

JS1

Leaves direction of joystick in port 1 on stack.

JS2

Leaves direction of joystick in port 2 on stack.

n KB

Leaves true value on stack if key number n is pressed.

LCASE

Go into lower case.

LORES

Go into LORES mode.

LPX

Leaves light-pen X-position on stack.

LPY

Leaves light-pen Y-position on stack.

MAR Parameters: SPN,COL,ROW,WID,HGT

Mirror attributes left-to-right in window.

" filename" MERGE

Load in sprites, keeping existing ones.

MIR Parameters: SPN,COL,ROW,WID,HGT

Mirror data left-to-right in window.

n MODE

Sets mode number for PLOT, POLY, BOX etc.:

0 to 3 - colour to be plotted in multi-colour code.
0 or 1 - clear point in 2 colour mode.
2 or 3 - set point in 2 colour mode.
4 - invert point.

MONO

Puts hardware into 2-colour mode.

MOVAND Paramaters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

W1 AND W2 -> W2

MOVATT Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2 (move attributes only)
MOVBLK Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2 W1 -> W2
MOVOR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2 W1 OR W2 -> W2
MOVXOR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2 W1 XOR W2 -> W2

flag MUTE

Enables/disables muting of voice 3.

length voice MUSIC

Sound note; length in 60ths of a second.

voice NOISE

Set up voice to generate noise.

MULTI

Puts hardware into 4-colour mode.

```
OR%AND          Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
OR%BLK          Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
OR%OR           Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
OR%XOR          Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
```

OSC

Leaves output from voice 3 oscillator on stack.

n PASS

Set filter to low pass (n=0), high pass (1), band pass (2), or notch reject (3).

PLOT Parameters: SPN,COL,ROW

Plot a point.

POINT Parameters: SPN,COL,ROW

Leave value of point referenced on stack:

0 or 1 if in S2COL mode.
0, 1, 2, or 3 if in S4COL mode.

POLY Parameters: SPN,COL,ROW,WID,HGT,NUM,INC

Draw polygon

width voice PULSE

Set voice to generate pulse wave form; 0<width<4096.

PUTAND Parameters: SPN,COL,ROW

Move sprite to screen at (COL,ROW), ANDing with screen.

PUTBLK Parameters: SPN,COL2,ROW2 move directly to screen.
PUTOR Parameters: SPN,COL2,ROW2 OR with screen.
PUTXOR Parameters: SPN,COL2,ROW2 XOR with screen.

PUTCHR Parameters: SPN,COL,ROW,NUM

Copy character block from sprite into character memory. NUM same as for CHAR.

" filename" RECALL

Load in new sprites.

RESET

Erase all sprites and reset sprite storage to bottom of memory.

RESEQ

Renumber sprites.

n RESONANCE

Set resonance (0..15) of filter.

flag voice RING

Enable/disable ring modulation.

S2COL

Puts software into 2-colour mode.

S4COL

Puts software into 4-colour mode.

voice SAW

Set voice to generate sawtooth waves.

SCAN Parameters: SPN,COL,ROW,WID,HGT

Leaves true value on stack if window contains data.

SCL1 Parameters: SPN,COL,ROW,WID,HGT

COMMAND: Scroll left by one pixel.

similarly: SCL2,SCL8,SCR1,SCR2,SCR8

SCLR Parameters: SPN,ATR

COMMAND: Clear sprite.

n SCRLX

Shift screen left/right by 0 to 7 pixels.

SCROLL Parameters: SPN,COL,ROW,WID,HGT,NUM

Scroll window vertically by NUM pixels. NUM>0 = up, NUM<0 = down.

n SCRLY

Shift screen up/down by 0 to 7 pixels.

SETA Parameters: SPN,COL,ROW,WID,HGT,ATR

Set attributes in window to ATR

SIDCLR

Reset sound chip.

SPIN Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

Rotate W1 ->W2 by 90 degrees clockwise.

SPRCONV Parameters: SPN,COL,ROW,SPN2

Convert software to hardware sprite.

SPRITE Parameters: SPN,WID,HGT

Create new sprite, changing SPND.

" filename" STORE

Save sprites.

SWAPATT Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

Swap attributes in windows.

flag voice SYNC

Enable/disable synchronisation of a voice

colour TBORDER

Set text border colour

colour TPAPER

Set text paper colour.

voice TRI

Set voice to generate triangular waves.

UCASE

Go into upper case.

n VOLUME

Set master volume (0..15).

WCLR Parameters: SPN,COL,ROW,WID,HGT,ATR

Clear window.

n WINDOW

Set up hires/text window.

WIPE Parameters: SPN

Remove sprite from sprite table, changing SPND.

WRAP Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2,NUM

Scroll window by NUM pixels with wrap.

WRL1 Parameters: SPN,COL,ROW,WID,HGT

Wrap window one pixel left.

Similarly - WRL2,WRL8,WRR1,WRR2,WRR8

XOR%AND Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
W1 XOR W2 -> W1
W1 AND W2 -> W2

XOR%BLK Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
W1 XOR W2 -> W1
W1 -> W2

XOR%OR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
W1 XOR W2 -> W1
W1 OR W2 -> W2

XOR%XOR Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2
W1 XOR W2 -> W1
W1 XOR W2 -> W2

XPANDX Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

Expand W1 -> W2 in X direction.

XPANDY Parameters: SPN,COL,ROW,WID,HGT,SPN2,COL2,ROW2

Expand W1 -> W2 in Y direction.

IDEAL ERROR MESSAGES

Errors detected from within a graphics word are treated separately from the other FORTH error messages. The following 6 messages are possible:

?NO ROOM ERROR

There is insufficient memory available for the sprites being loaded into memory or created using SPRITE or ISPRITE, or COLD# has been given nonsensical parameters.

?CORRUPTED SPRITE ERROR

The sprite table data has been corrupted.

?REDEF'D SPRITE ERROR

An attempt has been made to redefine a spite that already exists using SPRITE or ISPRITE

?NO SUCH SPRITE ERROR

A sprite which does not exist has been referenced.

?DELETE SPRITE ZERO ERROR

Sprite zero is the screen and therefore it cannot be deleted using WIPE or DSPRITE.

?OUT OF RANGE ERROR

A parameters of a word does not fall within the permitted range of values.

Tape 1 Side A

BASIC Lightning, Sprite Generator, Arcade Sprites, Demo Sprites.

Tape 1 Side B

BASIC Lightning (Turbo), Sprite Generator (Turbo), Arcade Sprites, Demo Sprites.

Tape 2 Side A

Demo (type SHIFT RUN/STOP to load and execute).

Tape 2 Side B

White Lightning (Turbo Version), White Lightning (non-Turbo).

WHITE LIGHTNING

Following the success of White Lightning on the Spectrum 48K and its extensive use by commercial games writers, we present White Lightning on the Commodore 64. All the features of the earlier release are included, but full use has also been made of the Commodore's more powerful hardware.

▶ As well as a full Fig Forth, White Lightning has more than 100 additional sound and graphics commands. Basic text can be freely intermixed with White Lightning.

▶ Programs can be written in Lightning Integer Forth or, for newcomers to the language, in BASIC/Lightning hybrids—no knowledge of machine code whatsoever is required.

▶ Interrupt driven routines. This means that any routine can be executed up to 60 times a second while the main program runs normally.

▶ Up to 255 software sprites with user selectable dimensions (up to 6 screens wide!) as well as the Commodore's own hardware sprites.

▶ Software sprites can be scrolled, spun, enlarged, mirrored and more.

▶ Completed White Lightning programs run independently of the package and you are free to market them with no constraints or costs.

▶ A full sprite development package is also supplied complete with a library of predefined characters ready for use in your own games.

▶ In addition to this, a multi-tasking BASIC with all the White Lightning graphics commands is also supplied to help you experiment before writing your own White Lightning game. This is also available separately.

▶ A full demo and comprehensive manual make this without doubt the most powerful games writing system ever produced for the Commodore 64.

WHITE LIGHTNING IS JUST ONE IN A SERIES OF LIGHTNING PACKAGES FROM OASIS SOFTWARE.