



Dragon Data Ltd.

DRAGON FORTH

DRAGON FORTH

AO K30798

CONTENTS

PREFACE	2
OPERATING INSTRUCTIONS.....	3
INTRODUCTION TO DRAGONFORTH.....	4
INPUT/OUTPUT OPERATORS.....	4
MATHEMATICAL OPERATORS.....	6
STACK OPERATORS.....	9
OTHER OPERATORS.....	10
COLON DEFINITIONS.....	11
CONTROL STRUCTURES.....	12
CONSTANTS AND VARIABLES.....	16
USING THE EDITOR.....	18
ACCESSING BASIC.....	20
FORTH GRAPHICS COMMANDS.....	23
ERROR MESSAGES.....	26
FIG-FORTH GLOSSARY.....	28
ADDITIONAL GLOSSARY.....	52
EXAMPLE PROGRAM.....	54

DRAGON 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. 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.

DRAGON FORTH consists of a standard Fig-FORTH model, but with several extensions to the standard vocabulary of FORTH words. By far the most important extension to DRAGON FORTH is the ability to access almost ALL of the Dragon's own BASIC commands just as you would when writing a BASIC program, and the addition of many of the high resolution graphics commands as standard DRAGON FORTH words to speed up even further those programs requiring graphics. There is tremendous scope for writing programs harnessing the power of the graphics commands (CIRCLE, DRAW, PAINT, PLAY etc.) with the incredible execution speeds of DRAGON FORTH - the possibilities are limitless.

In addition to the basic vocabulary of DRAGON FORTH 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 - try doing that in BASIC!!!

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

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

DRAGON FORTH is specifically configured to make maximum use of the capabilities of the Dragon. It is situated in the upper half of memory (\$3600 onwards) in order to allow you to use the extra HIRES graphics pages (\$1800 to \$3600) or indeed to write a small BASIC program.

A line editor is included in DRAGON FORTH to enable you to write your own DRAGON FORTH source code for later compilation. Do not allow lines to exceed 32 characters, any characters 'wrapping round' are ignored. This source code is stored in memory at \$2200 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 DRAGON FORTH 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).

The program 'Dragon Forth' was written by Stuart Smith and is an enhancement of a program written by the Forth interest group to whom we offer our thanks.

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

Operating Instructions

1 Switch on the computer.

2 Rewind tape to the beginning.

3 Key in CLOADM"FORTH"
press play on the tape and wait for the load to complete.

4 If you DO NOT wish to load in the demonstration, then stop the tape and key in EXEC to run DRAGON FORTH.

5 If you DO wish to load the demonstration, turn the tape over rewind to the start of 'DEMO' then key in CLOADM"DEMO" and wait for the load to be completed. Key in EXEC to run DRAGON FORTH and the words 'DRAGON FORTH VERS 2.1' will appear.

Now to run the demonstration key in 4 LOAD and the words

```
"PAGE 1"  
"PAGE 2 OF DEMO"  
.  
.  
.  
.  
.  
.  
.  
"FINAL PAGE"
```

will appear, and the demonstration will now run.

During the compiling an error message will be displayed. Do not worry this is due to a redefinition of a FORTH word in which case the previous definition is ignored until the new definition is 'Forgotten'.

In order to stop the demonstration, press the "BREAK" key when the words "DRAGON FORTH" are being printed on the screen. If you wish to resume the demonstration, key in DEMO.

NOTE

If you have not loaded the demonstration or have not created your own Forth source code using the editor then DO NOT use the word LOAD since the computer may hang up.

INTRODUCTION TO DRAGON 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

DRAGON FORTH 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 spelled wrongly, or the number entered is not valid in the current base, then an error message will be displayed:

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 DRAGON FORTH 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 DRAGON FORTH word. As mentioned before, DRAGON FORTH (as do 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`

will instruct the FORTH computer to move into hexadecimal mode, push \$41 onto the stack, and then take that number and display it on the terminal - in this example the character displayed will be will be an "A". The actual character displayed may be any of the recognisable ASCII characters, a Dragon 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 then 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 \$41 (where the '\$' implies Hexadecimal 41 ie 65 decimal) onto the top of the stack. In order to display this character try the following example:

KEY (you will need to press a key) EMIT

CR : This will transmit a carriage return and a 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 16 and BASE is decimal (10), then . will print 22 (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 carriage return)
this will print 1F7

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

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

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

Suppose the top of the stack contains FF40, location FF40/41, contains 0014, 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 364F 20 TYPE

(Note that \$364F is pushed onto the stack and \$20 is pushed on top of it \$20 = TOP; \$364F = second). This will print \$20 (32) ASCII characters corresponding to the data starting at address \$364F . (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 its values from the stack in the same way as for TYPE, i.e. top = character count, second = starting address. The output this time will consist of a dump in Hexadecimal of the required area in groups of 16 bytes, followed by the ASCII character equivalents.

e.g. : HEX 364F 20 DUMP
will produce

37 06 ED 84 20 22 EC 84 followed by characters represented by these codes.

." : 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 "
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
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 + .

will print the value of $1 + 2 = 3$ on the screen. Note that the two top numbers are removed from the stack and 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 next examples let us refer to the numbers on the stacks 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:

01FA 0019 1F47

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>

we get 1F47
19
1FA

We will now resume our explanation of the mathematical operators.

- : This will subtract the second number on the stack from the top number on the stack and leave the result as the top number.

i.e. : N1 = N2 - N1

e.g. : 7 11 - .
will print -4, since the stack would contain

N1 11 ← TOS (Top of stack)
N2 7

before the subtraction, and

-4 ← TOS

after the subtraction.

* : This will multiply the top two numbers on the stack and leave the result.

i.e. : N1 = N1 x N2

e.g. DECIMAL 140 20 * .
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 / .
will print 2

MAX : This will leave the greater of the number at the top of the stack and the second number on the stack .

e.g. : 371 309 MAX .
will print 371

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

e.g. : 371 309 MIN .
will print 309

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

e.g. : 47 ABS .
will print 47

-47 ABS .
will print 47

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

e.g. : 418 MINUS .
will print -418

-418 MINUS .
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- .
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 .
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 .
 will print 5 (quotient)
 2 (remainder)

3. STACK OPERATORS

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

e.g. : 719 DUP . .
 will print 719 719

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

e.g. : 111 222 DROP .
 will print 111

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

e.g. : 111 222 SWAP . .
 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 .
 will print 111
 222
 111

since the stack before OVER was

222 ← TOS
 111

and after OVER is

copy → 111 ← TOS
 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 .
 will print 1
 3
 2

since the stack before ROT is:

3 ← TOS
 2
 1

and after ROT

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:

3000 ← TOS
FFEE

This will store FFEE at address 3000/3001

i.e. FF at 3000

EE at 3001

If we key in FF00 2222 !

this will store FF00 at 2222/2223

i.e. 2222 contains high byte FF

2223 contains low byte 00

Remember that each 16 bit number takes up 2 bytes.

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

Suppose the memory contents are as follows:

Address:	3600	3601	3602	3603	3604	3605
Contents:	16	01	3F	16	01	8E

then 3600 @ .

will print 1601

If we 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 3600 C!

will store a single byte FF at address 3600.

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.

e.g. : refer to memory contents shown previously.

If we key-in

3600 C@.

this will print FF (and not FF01 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 3600 +!

will add 4 to the value at 3600/3601

As will be shown later, this is of use when using variables in DRAGON FORTH.

5. COLON DEFINITIONS

These are the most powerful and most used forms of data structures in DRAGON FORTH, 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 90 characters.

The general format is:

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

All colon definitions end with a semi-colon ;.

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

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

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 * . . ;
```

Here we have defined a new word called SQUARE which will be called by number SQUARE (CR)

e.g. 9 SQUARE

will result in

THE SQUARE OF 9 IS 81

If we follow the operation of the word, we 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 realised that the control structures can only be incorporated in colon definitions else an error will be displayed.

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.

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

```
: TEST2 5 0 DO ."HELLO" 2 +LOOP ;
and 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 is a word to do this:

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

```
: TEST3 0 10 DO I . -2 +LOOP ;
will print
10 8 6 4 2
```

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<
will leave a true flag (non-zero).

To see this, type

to print the top number on the stack, which is the flag. This will print
1
to show a true flag.

914 0< .
will print 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 < .
will print 0 (false flag).

If we look at the stack during this operation we 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 > .
will print 1 (true flag).

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

Now for some examples using the conditional branching structures:

```
: TEST= = IF ." BOTH ARE EQUAL " ENDIF ." FINISHED " ;
```

Now key in 2 numbers followed by TEST= and a carriage return.

e.g. 11 119 TEST=
This will print FINISHED

119 119 TEST=
will print BOTH ARE EQUAL FINISHED

```
: TEST1= = IF ." EQUAL " ELSE ." UNEQUAL " ENDIF CR ." FINISHED " ;
```

Now key in
249 249 TEST1=
this will print EQUAL
 FINISHED

Try 249 248 TEST1=
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 the following example:

```
: COUNT-DOWN DECIMAL
```

```
100 BEGIN 1- DUP DUP . CR  
0= UNTIL ." DONE " ;
```

This will print:

```
100  
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 the top of the stack and if that flag is true, then 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+ DUP
DUP DUP DUP * *
DUP 3000 <
WHILE ." THE CUBE OF "
SWAP . ." IS " . CR
REPEAT
DROP DROP DROP
." ALL DONE " CR ;
```

You may get an error message "err#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 still 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

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)
TOS ↓ empty	DECIMAL	
0	0	
0	BEGIN	
1	1+	

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

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

If TRUE:

N N N ³	WHILE	
N N ³ N	."	THE CUBE OF "
N N ³	SWAP	THE CUBE OF
	.	N
N	."	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 "
	CR	ALL DONE
	;	

In fact, it is a good idea to check the stack contents during the execution of any new DRAGON FORTH word to make sure that it is working correctly.

CONSTANTS AND VARIABLES

DRAGON FORTH 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 and the DRAGON 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 reference 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 P
1000 CONSTANT Q
256 VARIABLE X
0 VARIABLE Y
```

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

OTHER COMMONLY USED DRAGON FORTH WORDS

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

e.g. 5 LIST will list screen 5 on the screen. At the moment the screens 8 and upwards will contain garbage since no editing has been carried out. If you have already run the demonstration program then screens 4-7 will contain the source code for this, otherwise they too will contain garbage.

FORGET : this is used to delete part of the DRAGON FORTH dictionary. 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 the BREAK key will stop the listing.

Note: DLI, DLO, GO, BL, BLOCK-READ, BLOCK-WRITE, DLIST, INDEX and TRIAD do not apply to DRAGONFORTH and should not be used.

LOAD : this will compile the source code that you have created using the editor into the DRAGON FORTH dictionary to become new DRAGON FORTH words. Loading will terminate at the end of a screen or at the DRAGON 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

Included in this version of DRAGON FORTH is a line editor to enable you to create source or text files. To facilitate text editing, the text is organised into blocks of 256 bytes, divided into 8 lines of 32 characters to enable it to fit neatly onto a single DRAGON display screen. Once the text has been edited, it may then be compiled into the DRAGON FORTH dictionary and the text can be saved to tape. The text is stored in memory in the extra graphics pages at \$2200 and \$3600 therefore you can edit into screens 4 to 27. If during a program the extra graphics pages are going to be needed for graphics or for storing BASIC variables such as large arrays, then the text will need to be saved to tape otherwise it will be overwritten and lost.

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, move all the other lines down by 1 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.

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 at 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.

CLEAR : Clear the screen number held on the stack, and make it the current screen.

WHERE : If an error occurs during the loading of DRAGON FORTH 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 this screen or you may move to another screen by either listing or clearing it.

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

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

LOAD

This will start loading at the screen number held on 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

;S

This means "stop loading and interpreting".

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

FLUSH

This allows you to save your text on tape using

BC CSAVEM "filename", &H2200, &H3600, &H3742]

There now follows an example on how to edit a text file:

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

9 CLEAR

This sets the current screen to 9. To insert text into this screen we use the word P:

```
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>
```

9 LIST will produce

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

ACCESSING BASIC

DRAGON FORTH is even more powerful than standard Forths in that it allows you to access nearly all the BASIC commands (except those which require line numbers such as GOTO GOSUB - see list further on)

Variables may be used as space is reserved for them in the extra graphics pages and string variables have 2K reserved for them at the top of memory.

The BASIC words are accessed in the following manner:

```
B[....Basic statements....]
with the BASIC words in between [and ]
([ is shift (arrow down) and ] is shift (right arrow)).
e.g.  B[ PRINT 12345 ]
```

Multiple statements may be put between the [and] if required.

The text data will be saved and loaded using the machine code save and load commands in BASIC:

```
To SAVE a text file use
B[ CSAVEM " filename ". &H2200, &H3600, &H3742 ]
```

```
To LOAD a text file use
B[ CLOADM " filename " ]
```

When handling strings in BASIC via the B[word it is necessary to take the following steps if handling strings in the 'Immediate' mode (i.e. outside a colon definition):

The string must be defined with a NULL string added in order to preserve it.

e.g. try :-

```
B[ A$="DRAGONFORTH" ]
B[ PRINT A$ ]
```

and you will see that the string has been lost.

Now Try :-

```
B[ A$="DRAGONFORTH"+" " ]
B[ PRINT A$ ]
```

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[ BS="DRAGONFORTH" ];
STRING
B[ PRINT BS ]
```

and the string is OK.

Run the demonstration program and list screens 4-18 to see more examples of DRAGON FORTH.

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

```
e.g. 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 inputting text in the "immediate" mode, you may type in up to 90 characters before pressing carriage return.

If an error is detected during the execution of a BASIC command then an error message (as in BASIC) will be output and control will return to the "immediate" mode of DRAGON FORTH.

If you are using the high-resolution graphics mode, it is necessary to issue a BASIC 'PRINT' command in order to return to the text input mode, in fact it may well be worthwhile to define a DRAGON FORTH word to do this.

```
: TEXT B[ PRINT ] ;
```

Another useful word to define is a word to enter the graphics mode. For example, to enter mode 3 -

```
: GR B[ PMODE3 : SCREEN1,0 : PCLS ] ;
```

Try this example:

Firstly, define the words TEXT and GR as above (an error message 4 will be printed in relation to TEXT but this will not cause problems).

Then

```
GR B[ CIRCLE (100,100), 75, 3: PAINT (100,100)), 3, 3 : PAINT (1,1), 2, 3 ]
```

Since you are now in the high-resolution graphics mode you cannot see what you are keying in, so carefully key in

```
TEXT
```

to return to the text input mode.

Although the power and versatility of DRAGON FORTH is enhanced by the ability to interface to BASIC, there are some limitations involved in this interface:

LIMITATION TO BASIC INTERFACE

In general, any commands which involve line numbers or cannot be used in the immediate mode of BASIC will generate errors in DRAGON FORTH.

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

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

The following commands give errors (ID) when executed in the immediate mode and therefore cannot be used:

```
INPUT
DEF
READ
DATA
```

This next list of commands are SYSTEM commands which require a BASIC program to operate on, and therefore cannot be used in DRAGON FORTH

```
LIST
EDIT
CONT
NEW
RENUM
RUN
DEL
TRON
TROFF
```

Other commands to be avoided are:

STOP and END - since these return control to BASIC

FOR ... NEXT - this will execute correctly, but returns with a SYNTAX error on completion.

DIM should be used with great care as there is only limited space available for variables in BASIC

1K for numeric variables
2K for string variables

However, it is perfectly safe to use all the GRAPHICS commands and all the MUSIC/SOUND commands.

FORTH GRAPHICS COMMANDS

In addition to the BASIC graphic commands which can be accessed via the BASIC interface, many of the high-resolution graphics commands have been implemented as standard DRAGON FORTH words to enable even faster manipulation of the graphics screens.

FORTH WORD	BASIC EQUIVALENT
PPOINT	PPOINT
PRESET	PRESET
PSET	PSET
LINE,S	LINE,PSET
LINE,S,B	LINE,PSET,B
LINE,S,BF	LINE,PSET,BF
LINE,R	LINE,PRESET
LINE,R,B	LINE,PRESET,B
LINE,R,BF	LINE,PRESET,BF
PUT	PUT
GET	GET
GET,G	GET(X,Y)-(X,Y),array,G
PUT,S	PUT()-(),,PSET
PUT,R	PUT()-(),,PRESET
PUT,AND	PUT()-(),,AND
PUT,OR	PUT()-(),,OR
PUT,NOT	PUT()-(),,NOT
GETKEY	INKEY\$

There is an additional word called SHAPE defined in DRAGON FORTH which allows the user to manipulate graphics shapes by a name via the PUT and GET commands and their options.

There now follows a more detailed account of the DRAGON FORTH graphics words.

SHAPE: this word allows you to reserve space for a shape to be manipulated using PUT and GET (with options) in much the same way as an array is used in BASIC. The length and height of the area to be saved are used to reserve the space.

The general form is:

length (x_2-x_1) height (y_2-y_1) SHAPE shape-name

e.g. 20 10 SHAPE ROCKET

will reserve a space for a shape named ROCKET which has a length of 20 and a height of 10.

The SHAPE command must be used to define a shape and reserve space for it, before it can be manipulated by PUT and GET.

GET: this allows you to get an area of screen memory into the area defined by SHAPE.

The general form is:

X-coordinate Y-coordinate shape-name GET

where X-coordinate and Y-coordinate are at the top left had corner of the screen area to be got.

Shape-name is the name of a previously defined shape using SHAPE.

It is not necessary to specify the bottom right corner since this is calculated by DRAGON FORTH from the length and height used in defining the shape by SHAPE.

ALL the PUT and GET commands, including their various options, i.e. PUT,OR ; PUT, PSET etc., use the form:

X-coordinate Y-coordinate shape-name ---

where --- are any of the PUT/GET commands.

PUT: allows you to put data onto the screen from a previously defined shape. This can be used only if GET was previously used.

e.g. 20 10 SHAPE ROCKET
100 100 ROCKET GET
150 150 ROCKET PUT

will reserve space for a shape called rocket which will be of length 20 and height 10. Data will be got from the area of the screen (100,100)-(120,110) into the area reserved for ROCKET.

This data will then be put back on the screen at coordinates (150,150)-(170,160).

GET,G: this does the same as GET except that it allows the use of the "PUT, option" form of the PUT command.

PUT,S

PUT,R

PUT,AND

PUT,OR

PUT,NOT : these can only be used if the GET,G option was previouslyy with the shape.

e.g. 20 10 SHAPE MISSILE
50 50 MISSILE GET,G
100 100 MISSILE PUT,S

This will reserve space for a shape called MISSILE.

Get data from screen (50,50)-(70,60) into the shape and allow use of PUT options.

Put data onto the screen from the shape in the same way as it was taken from the screen.

PUT,S; PUT,R; PUT,AND; PUT,OR; PUT,NOT operate on the screen data in the same way as their BASIC equivalents.

PSET: this will set a specific point to a specified colour.

X Y C PSET

e.g. 100 150 3 PSET

will set point (100,150) to colour 3.

PRESET: will reset a particular point to the background colour, i.e. turn it off.

X Y PRESET

e.g. 150 75 PRESET

will reset the point (150,75).

PPOINT: this tests a specific high resolution point and returns the colour code of that point if it is on, if it is off.

The colour code will be returned to the top of the stack.

X Y PPOINT

e.g. 100 150 PPOINT .

will print 3 if the example shown in PSET was carried out.

GETKEY: this will test the keyboard and leave the ASCII value of the key pressed on the stack or zero on the stack if no key is pressed. This will not wait for a key to be pressed as in KEY.

All the various forms of the LINE command use the general form of:

X Y X Y ---

where X Y are the start coordinates of the line and X Y are the end coordinates of the line.

e.g. 100 100 150 150 LINE,S

will draw a line in the current foreground colour from (100,100) to (150,150).

If X and Y are entered as negative numbers (any negative numbers will do), then the start point is taken as the last end point of the LINE command.

e.g. -1 -1 200 180 LINE,S

will draw a line from (150,150) to (200,180) if the previous example was used.

Again, the DRAGON FORTH words operate in the same manner as their BASIC equivalents

FORTH ERROR MESSAGES

The following error messages may occur, and they will be printed out using the appropriate error number:

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

e.g. 109Z

err 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 ;  
      TEST1
```

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

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

```
Try: TEST2 1000 0 DO ?STACK 0 0 0 0 0 LOOP ;  
      TEST2
```

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

```
: TEXT B[ ? ] ;
```

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.

err 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 see the glossary (words with "C" in the top right hand corner of description).

```
Try: DO  
      IF
```

err 18 - this occurs if a word meant for execution only is put within a colon definition.

err 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 " ;
```

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

```
Try: TEST4 IF ." OK " ;
```

err 21 - this means that you have tried to delete something in the protected part of the DRAGON FORTH dictionary.

FORGET DO

err 22 - this implies the illegal use of → when not loading text screens.

→

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

Try: 12 D

There are a further two errors which occur when editing or loading or listing screens of data.

Range? - this means that you are trying to access a screen which is out of bounds of the memory. The screens must be in the range 4 to 23 inclusive.

Try 25 LIST

Drive? - this occurs when you try to access a non-existent disc drive. Again, note that DRAGON FORTH uses a simulation of disk in memory from \$2200-\$3600, and therefore no use should be made of the commands which specifically apply to disk. Do not use DR0, DR1 or GO.

Fig-FORTH GLOSSARY

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, PO 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 one 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. On 8 bit data bus computers, 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 unspecified.

! 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.

d1 --- d2 LO

Generate from a double number d1, the next ascii character which is placed in an output string. Result d2 is the quotient after division by BASE, and is maintained for further processing. Used between <# and #>. See #S.

#> d --- addr count LO

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

' --- 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.

The run-time procedure compiled by D0 which moves the loop control parameters to the return stack. See D0.

(FIND) addr 1 addr 2 --- pfa b tf (ok)
 addr 1 addr 2 --- ff (bad)

Searches the dictionary starting at the name field address addr 2, matching to the text at addr 1. 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 32 indicates the full line text length.

(LOOP) C2

The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr 1 --- d2 addr 2

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 at 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).

- n1 n2 --- diff LO

Leave the difference of n1-n2.

—> P,LO

Continue interpretation with the next disc 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, and 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 blank 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 from the disc 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.

```
/          nl  n2  ---  quot          LO
```

Leave the signed quotient of nl/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 a 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.

ORANCH 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.

: 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 6809 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.

?ERROR f n ---

Issue an error message number n, if the boolean flag is true.

?EXEC

Issue an error message if not executing.

?LOADING

Issue an error message if not loading.

?PAIRS n1 n2 ---

Issue an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

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

?TERMINAL --- f

Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation.

a 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-originate memory. n is with regard to computer address type (byte or word).

AND n1 n2 -- n2 LO

Leave the bitwise logical and of `n1` and `n2` as `n3`.

B/BUF --- n

This constant leaves the number of bytes per disc buffer, the byte count read from disc by BLOCK.

B/SCR --- n

This component leaves the number of blocks per editing screen. By convention, an editing screen is 1024 bytes organised as 16 lines of 64 characters each.

However in Dragonforth, an editing screen is 256 bytes organised as 8 lines of 32 characters each.

BACK addr ---

Calculate the backward branch offset from HERE to addr and compile into the next available dictionary memory address.

BASE --- addr

A user variable containing the current number base used for input and output conversion.

BEGIN --- addr n (compilation) P,LO

Occurs in a colon-definition in form:

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

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

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 transfered from disc to which ever 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,LC

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 is marked up as updated, it is written to the disc. The block is not read from the disc. The address left is the first cell within the buffer for data storage.

C! b addr ---

Store 8 bits at address.

C, b ---

Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.

C(a) addr --- b

Leave the 8 bit contents of memory address.

CFA pfa --- cfa

Convert the parameter field address of a definition to its code field address.

OMOVE from to count --

Move the specified quantity of bytes beginning at address 'from' to address 'to'. The contents of address from is moved first proceeding toward high memory.

COLD

The cold start procedure to adjust the 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 COMPILER executes, the execution address of the word following COMPILER 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` to 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
0	00000000		00000000	00000000
1	00000001		00000001	00000001
2	00000002		00000002	00000002
3	00000003		00000003	00000003
4	00000004		00000004	00000004
5	00000005		00000005	00000005
6	00000006		00000006	00000006
7	00000007		00000007	00000007
8	00000008		00000008	00000008
9	00000009		00000009	00000009
10	0000000A		0000000A	0000000A
11	0000000B		0000000B	0000000B
12	0000000C		0000000C	0000000C
13	0000000D		0000000D	0000000D
14	0000000E		0000000E	0000000E
15	0000000F		0000000F	0000000F
16	00000010		00000010	00000010
17	00000011		00000011	00000011
18	00000012		00000012	00000012
19	00000013		00000013	00000013
20	00000014		00000014	00000014
21	00000015		00000015	00000015
22	00000016		00000016	00000016
23	00000017		00000017	00000017
24	00000018		00000018	00000018
25	00000019		00000019	00000019
26	0000001A		0000001A	0000001A
27	0000001B		0000001B	0000001B
28	0000001C		0000001C	0000001C
29	0000001D		0000001D	0000001D
30	0000001E		0000001E	0000001E
31	0000001F		0000001F	0000001F
32	00000020		00000020	00000020
33	00000021		00000021	00000021
34	00000022		00000022	00000022
35	00000023		00000023	00000023
36	00000024		00000024	00000024
37	00000025		00000025	00000025
38	00000026		00000026	00000026
39	00000027		00000027	00000027
40	00000028		00000028	00000028
41	00000029		00000029	00000029
42	0000002A		0000002A	0000002A
43	0000002B		0000002B	0000002B
44	0000002C		0000002C	0000002C
45	0000002D		0000002D	0000002D
46	0000002E		0000002E	0000002E
47	0000002F		0000002F	0000002F
48	00000030		00000030	00000030
49	00000031		00000031	00000031
50	00000032		00000032	00000032
51	00000033		00000033	00000033
52	00000034		00000034	00000034
53	00000035		00000035	00000035
54	00000036		00000036	00000036
55	00000037		00000037	00000037
56	00000038		00000038	00000038
57	00000039		00000039	00000039
58	0000003A		0000003A	0000003A
59	0000003B		0000003B	0000003B
60	0000003C		0000003C	0000003C
61	0000003D		0000003D	0000003D
62	0000003E		0000003E	0000003E
63	0000003F		0000003F	0000003F
64	00000040		00000040	00000040
65	00			

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 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 words 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.

$$D_1 \quad d_1 \quad n \quad \dots \quad d_2$$

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.

DARS d --- ud

Leave the absolute value ud 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 form:

DO ... LOOP
DO ... +LOOP

At run time, DO begins a sequence with repetitive execution controlled by a loop limit nl 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 nl 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), leaves 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 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 n --- LO

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

DUP n --- n n 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 times 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.

At 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.

ENDIF addr n --- (compile) P,CO,LO

At run time, ENDF 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 ENDF. Both names are supported in fig-FORTH. See also IF and ELSE.

At compile-time, ENDF computes the forward branch offset from addr to HERE and stores it at addr. n is used for error tests.

ERASE addr n ---

Clear a region of memory from zero to 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 or the count of characters has been received. One or more nulls are added at the end of the text.

FENCE --- addr U

A user variable containing an address below which FORGETTING is trapped. To forget below this point the user must alter the contents of 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. 2% 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 being 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 preceeding 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.

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.

Leave the ascii value of the next terminal key struck.

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

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.

Convert the parameter field address of a dictionary definition to its link field address.

A constant leaving the address just above the highest memory available for a disc buffer. Usually this is the highest system memory.

Display the ascii text of screen n on the selected output device. SCR contains the screen number during and after this process.

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 →.

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

Occurs in a colon-definition in form:

 DO ... LOOP

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

At compile time, LOOP compiles (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 --- d

A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

M/ d n1 --- n2 n3

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.

M/MOD ud1 u2 --- u3 ud4

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 --- max LO

Leave the greater of two numbers.

MESSAGE n ---

Print on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc un-available).

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.

MON

Exit to the system monitor, leaving a re-entry to Forth, if possible.

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, 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.

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.

OR n1 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 n1 n2 --- n1 n2 n1 LO

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

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

Input 80 characters of text (or until a "return") from the operators terminal. Text is positioned at the address contained in TIB with IN set to zero.

LI

R --- n

R#	--	addr	U
----	----	------	---

```
R/W      addr  blk  ---
```

R> -- n LO

RO -- addr U

REPEAT addr n --- (compiling) P.C2

BEGIN ... WHILE ... REPEAT

At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.

At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr.
n is used for error testing.

ROT n1 n2 n3 -- n2 n3 n1 IO

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.

SWAP n1 n2 --- n2 n1 LO

Exchange the top two values on the stack.

TASK

A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

THEN P, ∞, 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  addr1  n  ---  addr2
```

Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If $\neq 1$, the motion is toward low memory. The addr2 resulting is address of the other end of the name.

```
TYPE      addr  count  --      LO
```

Transmit count characters from addr to the selected output device.

$$U^* \quad u_1 \quad u_2 \quad \cdots \quad u_d$$

Leave the unsigned double number product of two unsigned numbers.

$$U/ \quad u_d \quad u_1 \quad \cdots \quad u_2 \quad u_3$$

Leave the unsigned remainder u_2 and unsigned quotient u_3 from the unsigned double dividend u_d and unsigned divisor u_1 .

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 (OBRANCH) and an offset from HERE to addr. n is used for error tests.

ADDITIONAL GLOSSARY

B[

Used in the form B[Basic statements]

Will allow the user to execute a string of BASIC statenents (delimited by a trailing]). If executed outside a definition, then the statements will be executed immediately. If placed within a definition then the statements will be compiled in-line for later execution. (uses the words (EX) and (TOKENIZE)).

Columns --- addr

A variable containing the number of characters per line (32).

SHAPE

Used to define a shape (area of screen memory) and reserve space for it in the dictionary for subsequent manipulation by put and get. Used in the form :

 n1 n2 SHAPE cccc

where n1 is the length of the shape
 n2 is the height of the shape
and cccc is the name of the shape.

JOYSTK n1 --- n2

Returns the joystick position (0-63) on the top of the stack and must be scaled to fit the screen. n1 must be between 0 and 3 to indicate the joystick measurement required.

n1=0...Horizontal left joystick
 1...Vertical left joystick
 2...Horizontal right joystick
 3...Vertical right joystick

1- n1 --- n2

Decrement n1 by 1.

2- n1 --- n2

Decrement n1 by 2.

JSR addr ---

Jumps to subroutine at address on stack which must end with a 'return from subroutine' instruction.

J --- n

Used within nested DO ... LOOP's to copy the second loop index onto the stack.

K --- n

Used within nested DO ... LOOP's to copy the third loop index onto the stack.

WARM ---

This will perform a warm-start.

NOOP ---

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

NOTE: All references to disc in this documentation can be read as references to the disc simulation area in memory from \$2200 to \$3600 which are treated as a very limited disc capacity by DRAGON FORTH and do not in any way change the operation or description of any of the FORTH words defined in this documentation. Do not use DR0, DR1 or GO .

It is, however, necessary to load any data into this area using the BASIC interface

B[CLOADM " name "]

before anything can be done with it; and it is necessary to save this area to tape at the end of a session in order to obtain a permanent copy of the text (source code) using

FLUSH B[CSAVE " name ", &H2200, &H3600, &H8000]

-see the section on the EDITOR for the editing commands.

EXAMPLE PROGRAM

We shall now write a small program to illustrate the use of DRAGON FORTH. We shall start writing the program starting at screen 19.

This program will move a shape around the screen and bounce off the sides of the screen.

In order to write this program we need the following:

- (i) a word to plot the shape on the screen,
- (ii) a word to erase the shape,
- (iii) a word to check for movement out of bounds of the screen.

Let us first draw and define the shape to be moved as a square box on the screen, using the editor.

19 CLEAR

0 P (PAGE 1 OF PROGRAM)

1 P 10 10 SHAPE BOX B PMODES

2 P 100 100 110 110 LINE,S,BF

3 P 100 100 BOX GET,G

4 P 125 VARIABLE BOX-X

5 P 95 VARIABLE BOX-Y

6 P 4 VARIABLE X-OFFSET

7 P 2 VARIABLE Y-OFFSET —>

Now key in

19 LIST

and check the contents of the screen. We will now go through the screen line by line explaining what is happening -

line - comment, identifying the screen.

line 1 - defines and reserves space for a shape called BOX which is to have a length and height of 10. Then, puts high resolution graphics into mode 3.

line 2 - draw and fill in a box from (100,100) to (110,110).

line 3 - get the box into the shape named BOX.

line 4 - set up the initial X coordinate of the shape to be moved.

line 5 - set up the initial Y coordinate of the shape to be moved.

line 6 - set up the variable containing the distance the shape is to be moved in the X direction.

line 7 - set up the distance to be moved in the Y direction. Inform the computer to continue to the next screen.

So far, none of this text has been executed since we are still in the process of editing it.

We now need to edit the next screen.

```
20 CLEAR
```

```
0 P (PAGE 2 OF PROGRAM)
```

```
1 P : ERASE-BOX BOX-X @ BOX-Y @
```

```
2 P BOX PUT,R ;
```

```
3 P : PLACE-BOX BOX-X @ BOX-Y @
```

```
4 P BOX PUT,S ;
```

```
5 P : UPDATE-POSITION BOX-X @
```

```
6 P X-OFFSET @ + BOX-X ! BOX-Y @
```

```
7 P Y-OFFSET @ + BOX-Y ! ; -->
```

Again we will go through it line by line.

line 1/2 - this defines a word which will blank out the space at the X and Y coordinates given by BOX-X and BOX-Y.

line 3/4 - this defines a word to place a shape (defined by BOX) at the X,Y coordinates given by BOX-X and BOX-Y.

line 5/6/7 - this defines a word which will update the coordinates of the shape by adding the X offset to X and the Y offset to the Y coordinate. Then proceed to the next page.

Remember that if you call up a variable by its name, that its ADDRESS is pushed onto the stack and not its value. The value is manipulated by @ and !. Note that this program has not been optimised for speed or size but has been written as simply as possible.

Now onto the next screen.

```
21 CLEAR
```

```
0 P (PAGE 3 OF PROGRAM)
```

```
1 P CHECK-X BOX-X @ X-OFFSET @
```

```
2 P + DUP 5 < OVER 240 > OR SWAP
```

```
3 P DROP IF X-OFFSET @ MINUS
```

4 P X-OFFSET ! ENDIF ;

5 —>

Do the next screen.

22 CLEAR

0 P (PAGE 4 OF PROGRAM)

1 P : CHECK-Y BOX-Y @ Y-OFFSET @

2 P + DUP 5 < OVER 180 > OR SWAP

3 P DROP IF Y-OFFSET @ MINUS

4 P Y-OFFSET ! ENDIF ;

5 P —>

These two screens have defined words to check if the next X or Y coordinate will be out of bounds and if so will negate the offset to be added so that shape travels in the opposite direction.

We will discuss CHECK-Y in greater detail and look at its effect on the stack.

STACK	ACTION		RESULT
	TOS ↓ addr	BOX-X	put address of BOX-X on the stack.
	value @		put value on stack.
	value addr	X-OFFSET	put address of X-OFFSET on stack
	value value @		put value on stack.
	sum +		add offset to X and leave sum on stack.
	sum sum DUP		duplicate it.
sum	sum 5	5	push 5 on stack.
	sum flag <		check if sum < 5.
sum	flag sum OVER		copy sum to TOS.
sum	flag sum 240	240	push 240 on stack.
	sum flag flag >		check if sum > 240 i.e. off the screen.

sum	flag	OR	OR the two flags on the stack and leave the result. i.e. if either $X < 5$ or $X > 240$ then leave a single true flag else leave a false flag.
flag	sum	SWAP	swap top two values
	flag	DROP	drop the unwanted sum.
	empty	IF	test the flag left at top of stack.

If the flag is a 1 (true) then the following will be carried out, otherwise if the flag is a 0 (false) it will be skipped over .

True Part:

addr	X-OFFSET	put address on stack
value	@	get value
-value	MINUS	negate it
-value addr	X-OFFSET	put address on stack
empty	ENDIF	all done

By checking the stack during the execution of a word we can see if it does as is required.

Note that CHECK-Y works in exactly the same way.

We now have all the words we require and so can define a word to link them all together.

23 CLEAR

0 P (PAGE 5 OF PROGRAM)

1 P : GR B[PMODE3 : PCLS :

2 P SCREEN 1,1];

3 P START GR 2000 0 DO

4 P ERASE-BOX CHECK-X CHECK-Y

5 P UPDATE-POSITION PLACE-BOX

6 P LOOP ;

7 P ;S

This find screen defines a word to get into high resolution graphics and then defines a word which will go round in a loop 1000 times moving the shape around the screen.

Now all that remains is to compile these screens into the dictionary for execution.

Type in

```
19 LOAD
```

and wait for the "OK" to appear, then to run the program type

```
START
```

and sit back and watch. To get back to the text mode, key in B[print]

If an error occurs during the LOAD phase, key in

```
WHERE
```

to see where the error is and correct it and try again.

Now try changing the source code and see what happens. Key in the following:

```
FORGET TASK : TASK ;
```

to delete the compiled version.

We are just going to change line 4 of screen 20.

```
20 LIST
```

```
4 P BOX PUT,NOT ;
```

Now key in

```
FLUSH
```

```
19 LOAD
```

when the LOAD is completed and the OK appears, type

```
START
```

and watch the results.

HAPPY FORTHING!!!

Dragon Forth, in all machine readable formats and the written documentation accompanying them, are copyrighted. The purchase of Dragon Forth conveys to the purchaser a licence to use Dragon Forth for his/her own use and not for sale or free distribution to others. No other licence, expressed or implied, is granted.



Dragon Data Ltd.,
Kenfig Industrial Estate,
Margam,
Port Talbot,
West Glamorgan.
SA13 2PE.