SUPERFORTH
+
REVERSI

A COMPLETE FORTH-83 SYSTEM FOR THE QL

DIGITAL PRECISION

By
Gerry Jackson

Published by
Freddy Vachha

QL SUPERFORTH

CONTENTS

QL SUPERFORTH

(FORTH-83 Standard)

1 INTRODUCTION

1.1 SUPERFORTH is a standard FORTH-83 system with a complete set of double-number extensions (ie; 32 bit integer working as well as 16 bit integer working). The fundamentals of FORTH and all the features of SUPERFORTH are described. Unlike virtually every other FORTH software manual around, this one actually supplants the need for a separate FORTH reference guide. To learn and master FORTH you need nothing besides QL SUPERFORTH, this manual and your QL computer system.

Also included with the system are a Screen Editor for handling SUPERFORTH source code, a floating point maths package for doing non-integer arithmetic and a fully documented example game REVERSI, with 9 playing levels, which demonstrates the capabilities of SUPERFORTH.

SUPERFORTH uses the multitasking capabilities of the QL computer; it runs as a task so that other tasks can be run simultaneously. Also, SUPERFORTH words may be defined as tasks in their own right and executed simultaneously with other SUPERFORTH or machine code tasks.

Both in the Interpretive and in the Compiled modes, SUPERFORTH will execute considerably faster than SuperBASIC.

1.2 LOADING THE SYSTEM

After switch on or reset, SUPERFORTH is loaded by inserting the supplied cartridge into microdrive 1 and either pressing F1 or F2 or typing

```
LRUN MDV1_BOOT
```

When the system has finished loading, a message is printed: press CONTROL C (ie; hold down CTRL and press C) and a flashing cursor indicates that SUPERFORTH is waiting for input from the keyboard. As supplied, the Screen Editor will also be loaded using the Start-up command block facility described later, in section 6.

1.3 INPUT/ OUTPUT TO/FROM THE SUPERFORTH SYSTEM

As with SuperBASIC, input to SUPERFORTH can come from the keyboard or microdrives (or floppy disks) and output can be directed to the TV (or Monitor), microdrives or printer. Input to SUPERFORTH is in the form of "words" which are essentially strings of characters separated by spaces: a word can be a name, number, command or character and can contain any valid ASCII character.

Since SUPERFORTH is run as a task, keyboard input may be switched between SUPERFORTH and SuperBASIC at will by pressing Control C: eg; to list the directory of a microdrive.

2 SUPERFORTH FUNDAMENTALS

2.1 THE DICTIONARY

The SUPERFORTH system as supplied consists of a set of SUPERFORTH words pre-compiled into a dictionary. Using the SUPERFORTH system consists of either

(a) executing these pre-compiled words by typing them at the keyboard: this is using the SUPERFORTH system as an interpreter and is called Interpretive mode.

(b) compiling new words into the dictionary for later execution: this is usually called Compilation mode.

It is because new definitions can be compiled and then executed that SUPERFORTH systems are so much faster than SuperBASIC at executing programs. SuperBASIC always runs as an interpreter: ie; each line of the program always has to be analysed before it can be executed.

A SUPERFORTH program consists of a set of new definitions, compiled into the SUPERFORTH dictionary, which are executed by typing one word which calls the others as execution demands.

2.2 INTERPRETATION AND COMPILATION OF THE INPUT

When a word is entered in the input stream, ie; from the keyboard or microdrive, SUPERFORTH first of all searches for this word in the dictionary. If it is found then it is either executed (if in Interpretive mode) or compiled (if in Compilation mode). If it is not found in the dictionary, SUPERFORTH attempts to convert it into a number using the current number base. If this is not possible, an error is reported, and control returned to the keyboard, so that the user can correct the fault.

2.3 NUMBER HANDLING

All numbers, which have been entered, are treated as integers (see the floating point package description for real numbers), either 16 bit integers or double length 32 bit integers. The two are distinguished on input by inclusion of a decimal point in the number for a 32 bit integer: eg; 1234 is treated by SUPERFORTH as a 16 bit number, and 1.234 as a 32 bit number.

The point has no significance except to indicate that it is a double-number; however, the number of digits to the right of the point is stored in the variable DPL (see later) so that the user can implement real arithmetic if desired: eg; the numbers 12345., 1234.5, 1.2345 and .0012345 will all be treated as the double-number 12345, but DPL will contain the values 0, 1, 4 and 7 respectively.

Single length (16 bit) integers have a value in the range -32768 to +32767 if signed, or 0 to 65535 if unsigned. Double length (32 bit) integers have a value in the range -2147483648 to 2147483647 if signed or 0 to 4294967295 if unsigned. Negative numbers are preceded by a - sign without a space between the - and the number: eg; -123 . Positive assume that the stack contents are displayed in this way unless you are explicitly asked to type . or a similar word to output the TOS. You can revert back to the ok by typing

```
ASSIGN PROMPT TO-DO ok  <ENTER>
```

In subsequent descriptions of SUPERFORTH words you will come across the descriptor (n1 n2 --- n3) which represents the contents of the stack before and after the operation: ie; (before --- after), n2 being the TOS, n1 the 2OS and n3 the result of the operation. Hence the example above would be represented as (123 234 --- 357) for +

A more complex example might say

(n1 n2 n3 n4 --- n5 n6)

In this case, before the word is executed, n4 is the TOS, n3 the 2OS, n2 the 3OS and n1 the 4OS. After execution, n6 is the new TOS and n5 the new 2OS.

3 SUPERFORTH OPERATIONS

3.1 INTEGER ARITHMETIC OPERATIONS

First of all we will describe the integer arithmetic operations which operate on numbers held on the stack. SUPERFORTH words in this category are

+ - * /MOD / MOD NEGATE

which operate on 16 bit integers, and

D+ D- DNEGATE

which operate on double-length (32 bit) integers. Words that use double-length integers have d1,d2 etc in their stack description instead of n1,n2 etc.

More complex arithmetic operations will be considered in Section 3.3.

+ (n1 n2 --- n3) as described above. This adds the TOS n1 to the NOS n2 to give the sum n3: eg;

```
100 23 +
```

leaves 123 on the stack (if you have assigned PROMPT to do .S as previously described, you will see this). Type . to get rid of the 123, otherwise the stack will eventually fill up and an error message will result.

- (n1 n2 --- n3) subtracts n2 from n1 to leave the difference n3: eg;

```
100.23 -    leaves  77 on the stack
23 100 -    leaves -77 on the stack
```

* (n1 n2 --- n3) multiplies n1 by n2 to leave the product n3: eg;

```
123 3 *    leaves  369 on the stack
```

Note that the product is still a 16 bit integer. There are other multiplication words that will leave bigger products; these are described later.

/MOD (n1 n2 --- n3 n4) divides n1 by n2 to leave the quotient n4 and the remainder n3: eg;

```
10 7 /MOD   leaves 3 1 on the stack
```

The division is floored, which means that the quotient is always the nearest integer below or equal tothe true real quotient, and the remainder satisfies the equation

```
n1 = ( n2 * n4 ) + n3
```

This is true for both positive and negative numbers: eg;

```
-10  7 /MOD          gives n4=-2 and n3=4
 10 -7 /MOD          gives n4=-2 and n3=-4
-10 -7 /MOD          gives n4=1  and n3=-3
```

/ (n1 n2 --- n3) divides n1 by n2 to leave the quotient n3: eg;

```
120 30 /   leaves 4 on the stack
136 30 /   leaves 4 on the stack
```

Any remainder is lost.

and u3 = 12

ABS (n1 --- n2): n2 is the absolute value of n1, like the BASIC function ABS

```
eg;         123  ABS  gives  n2=123
           -123 ABS  gives  n2=123
```

DABS (d1 --- d2): a double-number equivalent of ABS

```
eg;        -123456. DABS D.   prints out  123456
```

MAX (n1 n2 --- n3) leaves the larger of n1 and n2 as n3

```
eg;          123  124 MAX        gives n3=124
             123 -124 MAX        gives n3=123
             124  123 MAX        gives n3=124
            -123 -124 MAX        gives n3=-123
```

DMAX (d1 d2 --- d3): a double-number equivalent of MAX

```
eg;        -123456. -123457. DMAX D.  prints  -123456
```

MIN (n1 n2 --- n3) leaves the smaller of n1 and n2 as n3

```
eg;          123  124 MIN        gives n3=123
             123 -124 MIN        gives n3=-124
            -124 -123 MIN        gives n3=-124
```

DMIN (d1 d2 --- d3): a double-number eqivalent of MIN

1+ (n --- n+1) adds 1 to the TOS. It is equivalent to the sequence 1 +

```
eg;          123 1+              gives   124
            -123 1+              gives  -122
```

1- (n --- n-1) subtracts 1 from the TOS

2+ (n --- n+2) adds 2 to the TOS

2- (n --- n-2) subtracts 2 from the TOS

2* (n1 --- n2) multiplies n1 by 2 to give n2. This is much faster than the equivalent 2 *

```
eg;         123 2*    gives n2=246
```

D2* (d1 --- d2): a double-number equivalent of 2*

2/ (n1 --- n2) divides n1 by 2 to give n2. Again, this is much faster than the equivalent 2 /

```
eg;         123 2/    gives n2=61
```

D2/ (d1 --- d2) a double-number equivalent of 2/

3.4 LOGICAL OPERATIONS

Four logical operations on numbers on the top of the stack are described here.

AND (un1 un2 --- un3): the bitwise logical AND of un1 and un2 is left as un3. This is useful for masking off unwanted bits in a number: eg; if we want to select the bottom 3 bits of 69 ,then 69 7 AND gives un3=5.

Also provided are double integer equivalents of some of these:

2DROP	(d ---): similar to DROP
2DUP	(d1 --- d1 d1): similar to DUP
2OVER	(d1 d2 --- d1 d2 d1): similar to OVER
2ROT	(d1 d2 d3 --- d2 d3 d1): similar to ROT
2SWAP	(d1 d2 --- d2 d1): similar to SWAP

3.6 CONDITIONAL TESTS

There are many words provided which compare numbers on the stack and leave a true or false result (usually called a flag) as the TOS. The two values of this flag are:

FALSE - represented by a zero
TRUE - represented by any non-zero value

The words described below always leave the TOS as a 0 for FALSE, and -1 (bits all 1s) for TRUE

Comparison operators are

<	(n1 n2 --- flag)	true if n1 < n2
=	(n1 n2 --- flag)	true if n1 = n2
>	(n1 n2 --- flag)	true if n1 > n2
<=	(n1 n2 --- flag)	true if n1 < n2 or n1 = n2
>=	(n1 n2 --- flag)	true if n1 > n2 or n1 = n2
<>	(n1 n2 --- flag)	true if n1 is not equal to n2
U<	(un1 un2 --- flag)	true if unsigned un1 < un2
U>	(un1 un2 --- flag)	true if unsigned un1 > un2
0<	(n1 --- flag)	true if n1 < 0
0=	(n1 --- flag)	true if n1 = 0
0>	(n1 --- flag)	true if n1 > 0
D<	(d1 d2 --- flag)	true if d1 < d2
D=	(d1 d1 --- flag)	true if d1 = d2
DU<	(ud1 ud2 --- flag)	true if unsigned ud1 < ud2
D0=	(d1 --- flag)	true if d1 = 0

examples are:

1 2 <	gives TOS = -1
-1 -2 <	gives TOS = 0
-2 1 <	gives TOS = -1
1 0>	gives TOS = -1
-1 0>	gives TOS = 0
1 2 U<	gives TOS = -1
-1 2 U<	gives TOS = 0 (because, as an unsigned number, -1 looks like 65535)

3.8 DEFINING NEW WORDS

Up to now, we have only typed in existing words to be executed immediately; this is SUPERFORTH working in its Interpretive mode. As in BASIC, programs can be stored for later execution. In SUPERFORTH this is achieved by compiling new definitions into the dictionary. However, in contrast to SuperBASIC, when the stored SUPERFORTH program is executed it runs very much faster because it has been compiled (in SuperBASIC the stored program is interpreted and so runs more slowly).

3.8.1 Colon definitions

The simplest method of compiling new word definitions into the dictionary is to use colon definitions, so called because the word : is executed.

```
eg;        : SQUARED DUP *  ;
```

compiles a word called SQUARED into the dictionary. This new word SQUARED can now be treated like any other SUPERFORTH word and can be executed or compiled. Note that it needs a number on the stack to square. Typing

```
           3 SQUARED           gives TOS =  9
and        11 SQUARED          gives TOS =  121
```

In the definition typed in above, the following actions occur:

(a)   : is executed to switch SUPERFORTH to Compile mode and to create a new dictionary entry. It takes the next word as the name of the definition.
(b)   DUP is the next word read in, but, since SUPERFORTH is now in Compile mode, it is compiled into the new definition SQUARED instead of being executed.
(c)   * is treated in the same way as DUP, ie; compiled
(d)   ; is then executed to terminate the definition and to switch SUPERFORTH back into Interpretive mode. (Note that ; is executed and not compiled, because it is a special word called an immediate word, of which more will be said later).

When SQUARED is executed, the words that were between SQUARED and ; are executed in turn, which has the effect of squaring the number on top of the stack.

: and ; must always occur in pairs and in that order. If ; is used without a preceding :, an error message will result. Colon definitions can be spread over more than one line: you will not, however, get the prompt ok in the middle of the definition.

Now SQUARED can be used in other definitions: eg;

```
           : TO_THE_FOURTH SQUARED SQUARED ;
then       2 TO_THE_FOURTH gives TOS = 16
```

A complete SUPERFORTH program consists of word definitions like these: the later ones use earlier ones as necessary and the final word runs the program: eg; in the accompanying game, REVERSI, there is a final definition called REVERSI which, when executed, causes the QL to play the game. A word must be defined before it can be used in the dictionary.

If you make a mistake in a definition, you can delete the whole word from the dictionary by using FORGET eg; typing

```
                    FORGET SQUARED
```

deletes SQUARED and any later words from the dictionary. If you now try to execute SQUARED you will get an error message. Note that TO_THE_FOURTH has also been deleted by the above command.

3.8.2 Switching between modes

Whilst in the middle of a colon definition you can switch SUPERFORTH between Compile and Interpretive modes using the words [and], which switch to Interpretive and to Compile mode respectively, eg;

```
          : TEST1 1 DUP + . ;           is simply compiled
          : TEST2 1 [ 123 . ] DUP + . ;
```

will compile the same actions but will print out 123 after you press ENTER

3.8.3 Immediate words

Some words are executed even if they occur in the middle of a colon definition. You have already met two of these: ; and [. Such words are called immediate words. If you want to make one of your definitions immediate, simply type the word IMMEDIATE after the definition, eg;

```
          : NOW 123 . ; IMMEDIATE
and       : TEST 1 2 + NOW . ;
```

will not compile NOW but execute it and print out 123 immediately. Typing TEST prints out 3 (don't forget to FORGET these words!).

Other words associated with DO ... LOOPs are (they must be used inside a DO ... LOOP or DO ... +LOOP):

I (--- n) leaves the value of the loop index on the stack, eg; ... 10 0 DO I . LOOP ... inside a colon definition will print out the numbers 0 to 9

J (--- n): like I, but leaves the next outer DO ... LOOP index. DO ... LOOPs can be nested, eg;

```
... 5 0 DO 2 0 DO J . LOOP LOOP ...
```

will print out the sequence 0 0 1 1 2 2 3 3 4 4

K (--- n): like I and J, except that it leaves the index of the second outer loop.

LEAVE This is used to prematurely terminate a DO ... LOOP. When LEAVE is executed, it branches to the word following LOOP or +LOOP, eg;

```
... 10 0 DO I . I 4 > IF LEAVE THEN LOOP ...
```

will print out the numbers 0 1 2 3 4 5

```
... 10 0 DO I 4 > IF LEAVE THEN I . LOOP ...
```

will print out 0 1 2 3 4

3.9.1 Case statement

To avoid the use of many IF statements, when a multiway decision is needed on the value on the top of the stack, a CASE statement is provided. An example of the use of CASE is (it can only be used in a colon definition):

```
: TEST    CASE 1 OF ." one" ENDOF
                2 OF ." two" ENDOF
               99 OF ." ninety nine" ENDOF
                  DEFAULT ." default"
          ENDCASE  ;
```

```
Now type  1  TEST    prints out  one
                     99 TEST   prints out  ninety nine
                     5  TEST   prints out  default
```

CASE .... ENDCASE (n ---) mark the start and end of'the statement.

OF (n1 n2 --- n1) tests n1 against n2. If equal, the words up to the next ENDOF are executed; if not, control passes to just beyond the next ENDOF.

ENDOF (---) marks the end of an OF ... ENDOF sequence.

DEFAULT (n ---) marks the start of the default sequence to be executed if none of the OF tests was equal.

U. (un ---): like ., except that un is printed as an unsigned number, eg;

```
 123 U.   prints 123
-123 U.   prints 65413
```

U.R (un n ---): like .R, except that un is printed as an unsigned number.

Words associated with these number-printing words are:

BASE (--- ad): a user variable that holds the current base for number conversion, both input and output. Use @ and ! in the usual way to read and load its value.

DECIMAL (---) loads decimal 10 into BASE

HEX (---) loads decimal 16 into BASE

Words that output characters and text to the screen are:

.((---): used in the form .(ccc...c) it prints out all the characters ccc...c, not including the space after .(
eg; .(Hello) prints Hello
It is most useful when compiling from microdrive or floppy disc.

." (---): like ., except that it prints characters up to a delimiting " and it can only be used in a colon definition. It compiles the message into the dictionary, eg;

```
: MESSAGE ." Hello there" ; MESSAGE
```

CR (---) outputs a new line (carriage return, line feed) so that subsequent output starts on a new line.

EMIT (n ---) outputs the least significant 8 bits of TOS as an ASCII character to the screen, eg;

```
65 EMIT   prints    A
66 EMIT   prints    B
43 EMIT   prints    +
```

SPACE (---) prints a space character

SPACES (n ---) prints n SPACE characters

TYPE (ad n ---) prints n characters from memory, where ad is the address of the first character (lowest address) in memory. This is normally used in conjunction with the next word COUNT .

Before COUNT is defined, we will look at how strings are stored in SUPERFORTH. They are stored as a sequence of characters, one per byte, in consecutive memory locations. In the byte before the first character in the string there is stored a count of the number of characters in the string, giving a maximum string buffer; the capacity of TIB is 85 bytes.

Now for multiple character input:

EXPECT (ad n ---) receives n characters or fewer if ENTER is pressed earlier, and stores them at address ad and consecutive higher addresses. All characters are displayed as entered and can be edited in the usual way. The number of characters is stored in variable SPAN .

QUERY (---) Up to 80 characters are read from the keyboard into TIB. The definition of QUERY includes the sequence TIB 85 EXPECT
#TIB is loaded with the number of characters. >IN and BLK (see section 4) are set to 0 .

Now a word to scan the characters input by EXPECT or QUERY:

WORD (n --- ad): n is the ASCII code for a delimiting character. WORD looks through the input stream (keyboard, microdrive or floppy disc) from the position indicated by the value of >IN , ignoring leading delimiters n, and transfers other characters into memory until the first trailing delimiter n is read. The characters accepted are stored as a counted string, as described above for screen output, at address ad . The value of >IN is adjusted to point to the character just beyond the delimiter. The action of WORD is terminated if the end of the input is reached. If this happens before any characters are read, the character count is zero. The stored string is followed by one space character, not included in the count.

CLS (---) clears the current output window. This is an execution vector (see section 8).

CSIZE (n1 n2 ---) sets the character size in the current output window to width n1 (range 0 to 3) and height n2 (0 or 1).

CURSOR (n1 n2 ---) positions the cursor using pixel coordinates relative to the top left corner of the window. n1 and n2 are the x and y coordinates respectively.

CURSOR_ON (---) switches the cursor on when expecting input.

CURSOR_OFF (---) switches the cursor off.

FLASH_ON (---) turns the flash state on. This only works in 8 colour mode (see MODE).

FLASH_OFF (---) turns the flash state off, eg;

```
8 MODE CLS
FLASH_ON ." Flashing" FLASH_OFF
```

INK (n ---) sets the colour of the ink in the current window to n .

MODE (n ---) sets the display mode for the QL to either 4 or 8 colour mode. Because of the way the QL works, this affects all windows and clears the screen (see SuperBASIC MODE).

PAPER (n ---) sets the colour of the paper in the current window to n .

STRIP (n ---) sets the colour of the strip in the current window to n,
eg; 7 PAPER 7 STRIP 0 INK
gives black ink on white paper.

PAN (n ---) pans the whole of the current output window n pixels, right if n is positive, left if negative.

PAN_LINE (n ---) pans the whole line containing the cursor by n pixels right or left as for PAN .

PAN_RLINE (n ---) pans the right hand end of the cursor line by n pixels right or left as for PAN .

SCROLL (n ---) scrolls the whole of the current output window by n pixels:
n positive scrolls downwards
n negative scrolls upwards.

SCROLL_TOP (n ---): like SCROLL, but only scrolls the top of the window, not including the cursor line.

SCROLL_BOTTOM (n ---): like SCROLL, but scrolls the bottom of the window, not including the cursor line.

X coordinate of centre
(see SuperBASIC CIRCLE)

FILL_ON (---) turns the graphics fill on (see SuperBASIC FILL)

FILL_OFF (---) turns the graphics fill off. Because of a limitation of the QL, you should always turn the fill off after drawing the shape, even if the next shape is to be filled.

LINE (ad ---) draws a line relative to the graphics origin. ad is the address of a list of 4 floating point parameters in this order:
Y coordinate of end of line
X coordinate of end of line
Y coordinate of start of line
X coordinate of start of line

POINT (ad ---) plots a point relative to the graphics origin. ad is the address of 2 floating point parameters in this order:
Y coordinate of point
X coordinate of point

RECOLOUR (ad ---): like SuperBASIC RECOL, this recolours the current output window: ad is the address of a list of 8 bytes which specify the new colours, in the order black, blue, red, magenta, green, cyan, yellow and white.

SCALE (ad ---) sets the scale and origin of the graphics coordinate system in the current output window. ad is the address of a list of 3 floating point parameters in this order:
Y coordinate of bottom line of window
X coordinate of lefthand pixel of window
length of Y axis
the default origin is (0,0) and the default height 100

Words to convert from ASCII strings to integers are:

CONVERT (d1 ad1 --- d2 ad2): the character string beginning at ad1+1 is converted and accumulated into d1 to give d2, by converting the string into digits, character by character until a non-convertible character is reached. As each charcter is converted, d1 is multiplied by BASE and the digit added to it. ad2 is the address of the first non-convertible character.

NUMBER (ad --- d) converts the count and character string at ad into double-number d using the value held in BASE. If conversion is not possible, an error message is printed. The string may contain a preceding minus sign, eg;

```
: CONVERT_AND_PRINT 32 WORD   ( read a word )
          NUMBER D.   ;       ( convert and print it )

CONVERT_AND_PRINT 123

: TEST_CONVERT
           32 WORD  ( read a word )
          CONVERT             ( convert it )
          C@ EMIT SPACE       ( print character )
          D.      ;           ( print double number )

0 0 TEST_CONVERT 4321765/     prints    / 4321765
10. TEST_CONVERT 123.         prints    . 10123
```

Another conversion word is:

S->D (n --- d) This converts a single integer to a double integer, the sign being retained.

ERASE (ad un ---): like BLANK, except that each byte is set to zero.

FILL (ad un n ---): un bytes of memory, starting at address ad upwards, are set to the least significant byte of n.

3.13 THE RETURN STACK

When a SUPERFORTH word is called from another SUPERFORTH word, the position to which program control will return when that word is completed is stored on a second stack called the return stack. While the contents of this return stack are usually of no concern to the programmer, it can be used with caution to temporarily save values from the main stack. Words handling this are as follows:

>R (n ---) transfers TOS to the return stack.

R> (--- n) transfers an integer from the return stack to TOS, removing n from the return stack.

R@ (---) reads the top of the return stack to TOS and leaves the value on the return stack.

There are restrictions on where and how these words are used. If misused, they are likely to crash the system. The rules are:

(a) >R, R> and R@ must only be compiled in a colon definition, never executed from the keyboard.
(b) inside a colon definition, >R and R> must occur <u>in that order</u> and must always occur in pairs as execution of that colon definition proceeds; ie; for every >R executed, an R> must be executed before ; or EXIT is executed.
(c) R@ must be used between >R and R> to be meaningful. As many R@s as needed can be used before R> .
(d) the >R and R> pair must not cross a DO... LOOP or +LOOP; ie; the pair must occur either outside or inside the loop. If they occur inside the loop, the SUPERFORTH words I, J and K must not be used between the >R R> pair, and the R> must occur before any LEAVE occurs.

An example of their use is a possible definition of ROT:

```
: ROT >R SWAP R> SWAP ;
```

4. MICRODRIVE AND FLOPPY DISK HANDLING

As well as compiling FORTH from the keyboard, SUPERFORTH will also compile FORTH source code stored on microdrive cartridges or floppy disks. As supplied, SUPERFORTH will use microdrives as the standard backing store: if you have made a backup copy (see 1.6) on floppy disk, the default will have been changed to floppy.

There are two ways of compiling from mass storage (as we will refer to microdrive or floppy disk from now on), one using standard SUPERFORTH blocks and the other using named files.

4.1 INPUT FROM STANDARD SUPERFORTH BLOCKS

The standard way of storing SUPERFORTH code on mass storage is to use standard size blocks of 1024 bytes: this method is used in SUPERFORTH. Each block is given a number in the range 1 to 65535 by the user and it is the responsibility of the user to keep track of the block numbers used. SUPERFORTH code or data can be entered into blocks using the supplied Screen Editor and saved to mass storage. Normally a SUPERFORTH program is stored in consecutively numbered blocks. When a block is used, SUPERFORTH reads it from mass storage into a block buffer: if it is changed in any way (for example by using the Editor), when another block is fetched from mass storage the new version of the first block is automatically saved and the original deleted.

SUPERFORTH decides whether input is to come from the keyboard or mass storage by examining the contents of the user variable BLK , which, if zero, defines input as coming from the keyboard, or, if non-zero, defines input as coming from the block number contained in BLK .

When a block is requested, SUPERFORTH first of all tries to read it from a default device, eg; MDV1_ . If it is not found there, SUPERFORTH then tries the other (of two) MDV or FLP, but does not change the default drive. If still not found, an error is reported on the display.

Because the operating system of the QL buffers all input/output to mass storage, in contrast to most FORTH systems only one block buffer is provided in the SUPERFORTH dictionary.

Blocks are stored on microdrive in small files named, for example, BLK123 for block 123, and, on floppy disk, FLP123 for block 123.

As usual, there are many words provided to handle mass storage using standard blocks:

B/BUF (--- n): a constant holding the number of bytes per buffer, this is set to 1024 and ought not to be changed.

BLK (--- ad): a variable holding the block number currently being used as the source of input. If BLK holds 0, input is taken from the keyboard (or a named file - see later).

UPDATE (---) marks the block buffer as having been updated, so that SAVE-BUFFERS, or the action of BLOCK and BUFFER, will save the buffer to the default mass storage device:
eg; UPDATE SAVE-BUFFERS ensures that a block is saved on the default device.

4.2 INPUT FROM NAMED FILES

Another way of inputting SUPERFORTH source code is from named files. This is not a standard FORTH method, but is convenient and fast when a program has been developed. The method makes use of the way input can be redirected in the QL to another channel. This method is used to load the Screen Editor, the game REVERSI and the floating point maths package.
Note that the word PROMPT is directed to execute no operation during the loading, otherwise the screen would fill up with'ok's. Also, since the input is being read one line at a time into TIB, no line in the named file should be longer than 85 characters; ie; there should be a carriage return or line feed character every 80 characters or fewer. Comments must not extend over more than one line.

Words provided to handle this are:

#FILE (--- ad): a double variable used to hold the double integer channel ID of the named file being used for input. It is loaded by LOAD_FILE . If for some reason the load fails, then the channel may be closed by:

```
          #FILE 2@ CLOSE
```

END_FILE (---) <u>must</u> be employed at the end of the file being used for input, to redirect the input stream to the keyboard and to close the channel.

LOAD_FILE (---) is used in the form

```
          LOAD_FILE MDV1_editor_fth
```

to, for example, load the Screen Editor. It redirects input from the named file and saves the channel ID in #FILE .

4.3 CREATING A NAMED FILE

There are two ways to do this:
(a) use the utility contained in block 4 to compact <u>consecutive</u> blocks into a file. To use this to save blocks 50 to 60 on the default mass storage device, in a file named example_fth on MDV1_, for example, type:

```
4 LOAD
50 60 SAVE_FILE MDV1_example_fth
```

Of course, any valid file name may be used on any device. You must ensure that END_FILE is included at the very end of the last block.

(b) use QUILL to generate the file. To do this you must use the

5. THE SCREEN EDITOR

The Editor is a full screen editor, which can be used to enter and edit standard SUPERFORTH blocks each containing 1024 characters of SUPERFORTH source code or data. Text is inserted simply by positioning the cursor and typing the required characters. Commands are available to edit the text and to assist in saving blocks on microdrives.

5.1 LOADING THE EDITOR

As supplied, the Editor is automatically loaded after SUPERFORTH is loaded. If this is changed, or if you have removed the Editor by using FORGET, it can be loaded by typing:

```
LOAD_FILE MDV1_EDITOR_FTH
```

5.2 ENTERING THE EDITOR

To edit an existing block, eg; block 678, type:

```
678 EDIT
```

which will enter the Editor and make the full range of commands described below available. To create and edit a new block, eg; block 932, type:

```
932 BUFFER DROP
932 EDIT
```

5.3 THE DISPLAY

The display in the Editor has three windows:
(a) at the top, the title and message window
(b) in the middle, the text window. This will display all 1024 characters of the block as 16 lines of 64 characters each. However, in the default mode, only 56 characters may be seen at any one time, the rest being seen by scrolling the display sideways. This happens automatically as the cursor is moved. This display may be redefined - see below.
(c) at the bottom, the line store window, which can be used to hold one or two lines temporarily as they are moved about a block or between blocks.

5.4 COMMANDS AVAILABLE

Various Editor commands are invoked by single or multiple key presses. If a displayable character key is pressed, it is inserted at the cursor position with the rest of that line and, optionally, the next line being moved to the right, the last character being lost from the end. In the description below the cursor control or arrow keys are called <left>, <right>, <up> and <down>. The command keys are as follows:

<left>,<right>,<up>,<down>
move the cursor around the screen.

CTRL <left>
deletes the character to the left of the cursor.

Note 1. If the current block has been modified in any way, it is saved to the default microdrive before the new block is read.

Note 2. You are asked for the block number in the top window. If you want to abort this command, pressing ENTER, without any other number, will return you to the old block.

Note 3. If you do not like the above choice of command keys , they can be redefined by editing the source code file of the Editor, using QUILL. You will need to import the file

editor_fth into QUILL.

Edit the key numbers in the large CASE statement in the word called EDIT, near the end of the file. Then print it to a file as described in section 4 (do it on a copy of the original, but be careful !).

5.5 MODIFYING THE DISPLAY

If you have a monitor for the display, which is capable of clearly displaying 80 characters per line, then you may wish to modify the default settings of the Editor windows and see the whole of the SUPERFORTH block at once, without the sideways scrolling. When the editor is loaded, the first thing it does is to load Block 3, which defines the windows and sets the display parameters. To define your own display, simply edit block 3 to define window sizes, colours, character sizes etc. The three display windows are called #1, #2, and #3, starting from the top. Change the constant C/D to 64 to see the whole 64 characters per line.
Again, edit a copy of Block 3 and be careful: editing the Editor can easily leave you without an editor !

If you wish to alter the number of characters per line to non-standard values, the following constants need to be changed:

C/L characters per line; default is 64

C/D characters per display; default is 56 or 64 (you must ensure that C/D is less than or equal to C/L)

L/B lines per block; default is 16

C/L and L/B are in the SUPERFORTH dictionary and can be changed either by using ' and >BODY or by redefining them in block 3.

6. SYSTEM INITIALISATION

6.1 STARTUP COMMAND BLOCK

After initialisation, or execution of COLD , SUPERFORTH loads and executes block 1, which is used to:

(a) define the console channel which is to be used as the default channel for the display and keyboard.
(b) define the paper,ink and strip colours for the default channel.
(c) define the character size to be printed in the default channel.
(d) do anything else the user cares to do; for example, as supplied, the Screen Editor is automatically loaded. This can save some lengthy typing-in of command sequences whenever SUPERFORTH is loaded.

This facility allows the user to select the display most suited to his requirements; for example, if he has a monitor, he will most probably want to define the option that gives him 85 characters a line, and employ a different set of colours to the user who has a TV display. To change the supplied settings, use the Screen Editor to modify block 1; it is best to modify a copy of block 1, to avoid accidents.

If block 1 is left blank or deleted, the system still loads correctly. If deleted, an error message will be displayed: simply ignore it. A new block 1 can be created using the editor, or by typing 1 BUFFER DROP UPDATE SAVE-BUFFERS.

6.2 SYSTEM RESTART

There are three ways of restarting the system, giving varying degrees of re-initialisation. These are defined with four words and another which clears the data stack.

ABORT (... ---) clears the data stack and performs the function of QUIT . ABORT is an execution vector, therefore the user may (with caution) redefine its action.

ABORT" (flag ---) is used in the form
ABORT" cccc"
so that, when it is executed, if the flag is true, then the message represented by characters cccc is displayed and ABORT executed. If the flag is false, the flag is dropped and execution continues.

COLD (... ---) completely re-initialises the system: the data and return stacks are cleared, the dictionary restored to the initial state and block 1 executed.

QUIT (---) clears the return stack, sets Interpretive mode and returns control to the keyboard.

SP! (---) clears the data stack.

7. ERROR HANDLING AND MESSAGES

There are many error conditions detected by SUPERFORTH. When these occur, the last word read from the input stream is output followed by a ?. A message is written to the display, execution aborted and control returned to the keyboard. The stack is left unchanged so that the user can possibly analyse the data held there to identify the cause of error. The messages output, their corresponding error number and their causes are now described.

7.1 ERROR MESSAGES

0 non-existent name or invalid number

when a word is not recognised and cannot be converted into a number using the current value of BASE .

1 Compilation mode only

when an attempt is made to execute a word while in Compilation mode, eg; ;

2 Execution mode only

when the system should be in Execution mode, eg; at the end of a mass storage block when control is returned to the keyboard.

3 control structure error

when an error is made in a control structure, eg; ... DO ... IF ... LOOP ... (ie; there is a missing THEN or ELSE ... THEN before LOOP).

4 stack mis-match in definition

at the start of a colon definition the depth of the stack is stored. When ; is executed, this value is compared to the current depth: if they differ, this message is output. This often detects a missing THEN in an IF statement.

5 use only when LOADing

when an attempt is made to execute the word --> from the keyboard or named file.

6 stack empty

whenever control is returned to the keyboard the depth of the stack is checked. If negative, this message is output and the stack pointer reset to the correct value.

7 stack full

as for stack empty, except that the stack is too full. There is room for 128 16 bit integers on the stack.

8 not found

when a word following ' or FIND is not found in the dictionary.

9 in protected dictionary

when an attempt is made to FORGET beyond the value of

ERROR (n ---) issues error message n and returns control to the keyboard.

If the user detects an error while his program is running, and wants to print out one of the error messages discussed above, this can be done by, eg;

```
             6 ERROR     to print 'stack empty' etc.
or          -4 ERROR     to print QDOS error 'Out of range' etc.
```

7.4 WARNINGS

One of two warnings may be issued. Since these may not result from error, but from circumstances intended by the user, no action results other than the issue of the warning itself. The two messages are:

10 redefining <name>
when a word called <name> already exists in the current vocabulary and is being superseded by another version.

22 Now in SUPERFORTH vocabulary
when FORGET has been used to forget past the top of the current vocabulary, SUPERFORTH detects this, tidies up the various linkages, selects the SUPERFORTH vocabulary and reports the fact.

You will have noted that the series of error and warning numbers have gaps: the missing numbers are used to print various system messages.

8. MORE ADVANCED TECHNIQUES

8.1 COMPILATION - ADDING TO THE DICTIONARY

We have already discussed some compiling words such as : ; CONSTANT and VARIABLE and their double length equivalents. Now we pursue the subject further to examine other ways of adding to the dictionary and other words associated with compilation.

[(---) sets Interpretive mode, usually within a colon definition (see LITERAL below for an example).

] (---) sets Compilation mode, usually within a colon definition (see LITERAL).

[COMPILE] (---) can only be used in Compilation mode and is used in the form

```
[COMPILE] <name>
```

to force compilation of <name>, which is the next word in the input stream. It is used to force compilation of an immediate word which would otherwise be executed instead of being compiled: eg; in a colon definition, the sequence ... [COMPILE] LITERAL would compile a call to LITERAL .

, (n ---) compiles the TOS into the next two available bytes in the dictionary.

C, (n ---) compiles the least significant byte of n into the next available byte in the dictionary.

ALLOT (n ---) allocates n bytes in the dictionary and updates the address of the next avilable location. The contents of the allotted bytes are undefined.

COMPILE (---) used in the form

```
: <name1> ... COMPILE <name2> ... ;
```

When <name1> is executed, COMPILE compiles the compilation address of <name2> instead of executing it; <name1> is usually immediate.

CREATE (---) is a defining word used in the form

```
CREATE <name>
```

to create an entry in the dictionary called <name> . When <name> is later executed, the address of <name> 's parameter field is left on the stack: eg; to CREATE a dictionary entry called FRED and to allocate 6 bytes to it, we type

```
CREATE FRED 6 ALLOT
```

Or, if we want to store a message in the form of a counted string,

```
CREATE MESSAGE 5 C, 72 C, 101 C, 108 C, 108 C, 111 C,
```

HERE (--- ad) leaves the address of the next available dictionary location.

IMMEDIATE (---) changes the last word defined in the dictionary into an immediate word.

LITERAL (n ---): when compiling, it compiles the TOS as a literal which, when the word being defined is later executed, will leave n as the TOS. It is often used in conjunction with [and] to do calculations in the middle of defining a new word , eg;

```
... [ 100 31 + 3 * ] LITERAL ...
```

will compile 393 as a literal. When the word containing this is executed, 393 will be left on the stack.
In fact, whenever you have used a number in a definition, it has been compiled as a literal without you realising it.

RECURSE (---) is used in Compilation mode only, to recursively compile the word currently being defined. This cannot be done by just typing the name, eg;

```
: CALLS_ITSELF DUP 0> IF DUP 1- CALLS_ITSELF THEN . ;
```

will not compile because CALLS_ITSELF does not exist in the dictionary until ; is executed, and so the compilation fails. However, replacing the second CALLS_ITSELF with RECURSE will give a word which prints an ascending list of numbers. Eg; try

```
5 CALLS_ITSELF
```

SMUDGE (---) is used either to enable or to disable recognition of the latest entry in the dictionary if it was previously disabled or enabled respectively: eg;

```
: TEST ;  SMUDGE TEST will work correctly
then      SMUDGE TEST   will not find TEST
and again SMUDGE TEST will work.
```

The normal use of SMUDGE is that when a new definition has failed to compile, it is left disabled (to prevent inadvertent execution of the word). The sequence SMUDGE FORGET <name> then FORGETs the faulty word.

```
eg;       : FAULTY IF ;          will not compile
          FORGET FAULTY          will not delete it
          SMUDGE FORGET FAULTY will delete it
```

STATE (--- ad) is a variable which defines the Compilation mode: STATE holds 0 when interpreting, and -1 when compiling.

8.3 DICTIONARY AND VOCABULARY MANAGEMENT

There is a series of words which allow you to manage and handle dictionary entries. You can search the dictionary for entries by name, and, perhaps most powerful of all, you can declare separate vocabularies of words.

8.3.1 Dictionary management

' (--- ad) is used in the form

```
' <name>
```

to search the dictionary for <name>. If <name> is found, then ad is the compilation address of <name> (ie; the address which is compiled into the dictionary when <name> occurs in a colon definition). If <name> is not found, error message 8 " not found " is displayed: eg;

' DUP U. prints the compilation address of DUP . See below for more examples.

['] (--- ad) is used in compilation mode only; it is used in the form

```
['] <name>
```

to search the dictionary for <name> . If <name> is found, then the compilation address of <name> is compiled into the dictionary as a literal, ie; when later executed, this compilation address is left on the stack (see LITERAL).

>BODY (ad1 --- ad2) converts the compilation address ad1 of a dictionary entry into a parameter field address (in fact, it is the same as 2+). A common use of this is to change the value of constants in conjunction with ' or ['], eg;

```
123 CONSTANT FRED FRED .       displays 123
456 ' FRED >BODY !             changes FRED
FRED .                         displays 456
```

['] can be used similarly, inside a colon definition.

EXECUTE (ad ---) executes the word whose compilation address is ad. If ad is not a valid compilation address, the system is very likely to crash: eg;

```
' DUP EXECUTE       does exactly the same as
DUP
```

FENCE (--- ad) is a user variable used to hold the address beyond which FORGET may not operate. It is used to protect against inadvertent deletions from the dictionary. If you want to protect some dictionary entries in this way, after compiling them type

```
HERE FENCE !
```

This protection can be cleared by changing the contents of FENCE suitably. If an attempt is made to FORGET beyond FENCE, error message 9 " in protected dictionary " is displayed.

FIND (ad1 --- ad2 n): like ', this is used to search the to execute a word which displays an error message.

SUPERFORTH (---) makes the SUPERFORTH vocabulary the vocabulary to be searched first of all. This is the primary vocabulary in which all the supplied words are situated and is, in fact, the only vocabulary until either the user defines a new one or the Editor is loaded. Note that this word is <u>NOT</u> immediate: previous FORTH standards, eg; FORTH 79, had FORTH as an immediate word; FORTH 83 does not.

FORTH-83 (---) ensures that a standard FORTH 83 system is available. If you FORGET past this word, you are very likely to crash the system. FENCE is initially set just past this word, to protect it.

VOCABULARY (---) is a defining word used in the form

```
VOCABULARY <name>
```

to define a new vocabulary which, when executed, will make <name> the first vocabulary to be searched when interpreting or compiling words.

An example of the use of vocabularies is:

```
FORTH DEFINITIONS   ( makes FORTH the compilation vocabulary and
            the first searched )
VOCABULARY SOCCER   ( creates a vocabulary named SOCCER )
VOCABULARY RUGBY    ( creates a vocabulary named RUGBY  )
SOCCER DEFINITIONS ( new dictionary entries now go in the
            SOCCER vocabulary )
: BALL ." is round" ;    ( defines the ball's shape )
: TEAMS ." have 11 men ; ( the number of players )

RUGBY DEFINITIONS  ( new entries go in the RUGBY vocabulary )

: BALL ." is oval"  ;     ( defines the ball's shape )
: TEAMS ." have 15 men ;  ( the number of players )
```

(note that you get no " redefining " warning messages).
Now type:

```
FORTH DEFINITIONS
BALL                ( gives an error message, as does TEAMS,
                      because they are not in the SUPERFORTH
                      vocabulary )
```

but now, typing:

```
SOCCER BALL          displays  is round
       TEAMS         displays  have 11 men
```

This is because typing SOCCER makes it the first vocabulary searched, so that SOCCER's definitions of BALL and TEAMS are found. Now try:

```
RUGBY BALL           displays   is oval
      TEAMS          displays   have 15 men
```

Now the RUGBY vocabulary is the first to be searched.

9. FLOATING POINT MATHS PACKAGE

A floating point package is provided in a separate file, which is not included in the main dictionary. This is because most applications do not need floating point facilities. The package is loaded by typing:

```
LOAD_FILE MDV1_FPMATHS_FTH
```

Words are provided to give a wide range of floating point maths operations using QDOS calls. The QL's floating point number format is used, which takes six bytes of memory for each floating point number. Where possible, the relevant integer word of FORTH 83 is preceded by an F, to give an equivalent operation on floating point numbers on the stack. Words provided are (fp refers to a six byte floating point number):

		FORTH 83 EQUIVALENT
FDUP	(fp --- fp fp)	DUP
FDROP	(fp ---)	DROP
FSWAP	(fp1 fp2 --- fp2 fp1)	SWAP
FOVER	(fp1 fp2 --- fp1 fp2 fp1)	OVER
F@	(ad --- fp)	@
F!	(fp ad ---)	!
F>R	(fp ---)	>R
FR>	(--- fp)	R>
FROT	(fp1 fp2 fp3 --- fp2 fp3 fp1)	ROT
FPICK	(fp...fp n --- fp...fp fp)	PICK
FROLL	(fp...fp n --- fp...fp)	ROLL
F0=	(fp --- flag)	0=
F0<	(fp --- flag)	0<
F0>	(fp --- flag)	0>
F<	(fp1 fp2 --- flag)	<
F>	(fp1 fp2 --- flag)	>
F=	(fp1 fp2 --- flag)	=
FCONSTANT	(fp ---) creates a floating point constant.	CONSTANT
FVARIABLE	(---) creates a floating point variable.	VARIABLE

Operations on floating point numbers are :

F+ (fp1 fp2 --- fp3)	does fp1+fp2 to give fp3
F- (fp1 fp2 --- fp3)	does fp1-fp2 to give fp3
F* (fp1 fp2 --- fp3)	does fp1*fp2 to give fp3
F/ (fp1 fp2 --- fp3)	does fp1/fp2 to give fp3
FABS (fp --- ¦fp¦)	similar to FORTH 83 ABS
FNEGATE (fp --- -fp)	similar to FORTH 83 NEGATE

10. SPECIAL QL FACILITIES

10.1 USE OF QL CHANNELS

The QL is able to direct input and output from/to any input or output device attached to the QL simply by using the appropriate channel number or channel ID. There are several SUPERFORTH words provided to handle this capability. First of all, there are some general channel handling words: you should note that the channel ID is a double length integer, so that 2@ and 2! should be used in conjunction with 2VARIABLEs to manipulate them. Also note the convention adopted of calling the 2VARIABLEs used to hold channel IDs by a name beginning with a # symbol, eg; #IN.

#DEFAULT (--- d): a double length constant used to hold the default channel ID, this is loaded by block 1 as supplied and is the channel ID loaded into #IN and #OUT whenever an error occurs. This ensures that a fault always returns control to the keyboard and display. #DEFAULT is loaded with a suitable value prior to block 1 being loaded.

#IN (--- ad): a double length user variable used to hold the channel ID of the current keyboard input stream (not the mass storage stream). By manipulating this, input may be obtained from other sources: this is the technique used by LOAD_FILE to load from a named file.

#OUT (--- ad): similar to #IN, except that it holds the channel ID of the current output device. Output may be redirected by manipulating this, which is the technique used to output to the printer.

CLOSE (d ---) closes the channel whose ID is on top of the stack. Always be careful to close channels when you have finished with them, to avoid profligate use of the QL's RAM.

OPEN (n --- d) is used in the form, for example,

0 OPEN CON_180X26A52X54

to open a console channel. The channel ID is left on top of the stack usually to be saved in a variable. Any valid device name, for screen windows, microdrives, floppy discs, serial interfaces etc can be used, but you must use the correct syntax as defined in the QL User Guide: eg;

0 OPEN MDV1_BLK99 (opens a channel to file BLK99)
0 OPEN SCR_180X26A52X182 (opens a screen window)

The parameter n used before OPEN is there primarily for microdrive files; it should be 0 for other devices. For microdrive files it should have the following values:

0	old (exclusive) file
1	old (shared) file
2	new (exclusive) file
4	directory

10.2 MULTI-TASKING

It is possible to multi-task both SUPERFORTH programs, which are compiled and created while SUPERFORTH is running, and machine code programs, which have been created independently of SUPERFORTH and stored on microdrive or floppy disc. Facilities are provided to create, activate, suspend and remove these tasks.

A SUPERFORTH task is provided on block 5: a clock display, which is used in examples below. To load, type

5 LOAD which loads but does not run the clock, for which see below.

10.2.1 Job identity

Whenever a task or job is activated on the QL it is allocated a double-number identifying it. This double-number is then used to manage the task. If a task wishes to refer to itself it can use a double-number -1 as the job identity. Two words are provided to utilise job identity:

?JOB_ID (--- d) is used in the form

?JOB_ID <name>

to find the job identity of a SUPERFORTH task created using JOB described below. It can not be used to find the identity of machine code tasks.

eg; ?JOB_ID CLOCK

JOB_ID (--- ad): a double variable which holds the identity of a machine code task which is activated using EXEC, see below. If you wish to manage this task you will probably need to save this value in another double variable.

eg; JOB_ID 2@ leaves the double number job identity in the stack after EXEC <name>

10.2.2 Creating tasks

The words available to create SUPERFORTH tasks are

JOB (ad n1 n2 n3 ---) : used in the form

JOB <name1> RUNS <name2>

to create a dictionary entry called <name1> which, when executed, will cause <name2> to be run as a multi-tasked program; <name2> must already exist. Eg; see block 2 for:

JOB CLOCK RUNS QLOCK

which creates a task called CLOCK which, when activated, will run a SUPERFORTH word called QLOCK. Note that the task is not yet activated: this must be done using START or ACTIVATE (see below).

ad is the address of the job's USER variables; ad=0 if there is none.

n1 is the number of long-words needed for the return stack (ie; n1*4 is the number of bytes)

n2 is the number of words needed for the data stack (ie; n2*2 is the number of bytes)

n3 is the job's priority (1 to 127)

suspends the clock for 10 seconds, after which it restarts. If n=-1 the suspension is indefinite.

SLEEP (---) is used by a task to suspend itself indefinitely by changing its priority to 0. This is compiled automatically at the end of a SUPERFORTH task by RUNS (to prevent a job "falling off the end").

SUSPEND (n ---) is used in the form

```
        SUSPEND <NAME>
```

to suspend task <name> for n fiftieths of a second.

```
Eg;     1000 SUSPEND CLOCK
```

suspends the clock for 20 seconds.

SUSPEND_ME (n ---) suspends the current task for n fiftieths of a second.

RELEASE (---) is used in the form

```
        RELEASE <name>
```

to restart <name>.

```
Eg;     -1 SUSPEND CLOCK     stops the clock
        RELEASE CLOCK        restarts it.
```

UNFREEZE (d ---) restarts the task whose identity is d .

10.2.5 Changing a task's priority

PRIORITY (d n ---) changes the priority of the task whose identity is d to n. n is in the range 127 (lowest) to 1 (highest). If n=0, the task is suspended.

PRIORITY_OF (n ---) is used in the form

```
        PRIORITY_OF <name>
```

to change the priority of task <name> to n. n has the same meaning as in PRIORITY, eg;

```
        25 PRIORITY_OF CLOCK
```

changes the clock's priority to 25.

10.2.6 Removing tasks

BYE (---) is used by a task to remove itself from the system. Typing in BYE removes SUPERFORTH from the system. You will need to press Control C to return to the SuperBASIC interpreter.

KILL (---) is used in the form

```
        KILL <name>
```

to stop and remove task <name> from the system. It must not be restarted by using START etc.

REMOVE (d ---) stops and removes the task whose identity is d from the system. Do not restart it.

SET_TIME (d ---) sets the time to double-number d seconds.

TIME (--- d) leaves the time on top of the stack as a double-number in seconds.

In addition to these, a utility block is provided to enable you to set the date and time. This is loaded and executed by typing

```
        2 LOAD
```

which, when loaded, requests the year etc. All replies must be integers (eg; MAY is month 5). The prompt and response sequence can be bypassed if you simply type ENTER in response to Year?

10.5 SERIAL INTERFACE/BAUD RATE

One word is included to adjust the baud rate of the RS232 interfaces:

BAUD (n ---) changes the baud rate to n
eg; 9600 BAUD changes it to 9600 baud

11. DETAILS OF SUPERFORTH IMPLEMENTATION

11.1 MEMORY MAP

DP SUPERFORTH uses over 68K bytes of memory. It is fully relocatable and the actual locations it occupies depend on what other tasks are running and whether extended RAM is fitted to the QL. The locations available to a standard SUPERFORTH program are 0 to 65535 relative to an absolute address held in register A2 of the 68008 microprocessor. The SUPERFORTH dictionary occupies locations 32768 upwards. Locations from (approximately) 42000 to 65535 and 0 to 31738 are available to the user, but the user can quite happily use the system without worrying about addresses (unless, of course, the space is completely filled up: which, given the compactness of SUPERFORTH, would imply a very big application indeed). The memory map is:

locations	32768 to 42000 (approximately)	SUPERFORTH dictionary
	42000 (approx.) to 65535	
and	0 to 31738	the user dictionary
	31739 to 32767	the block buffer

Some code and the error messages are situated outside the dictionary, to maximise the space available for the user.

11.2 THE STACKS

These are situated outside the dictionary area (for added protection) and just below the dictionary in the QL's memory. The return stack is immediately below the dictionary and there is room for 512 bytes, which is enough for 128 calls to SUPERFORTH secondaries. The data stack is just below the return stack and has room for 256 bytes or 128 integers.

11.3 DICTIONARY STRUCTURE

This information is supplied for the dedicated SUPERFORTH enthusiast, who is familiar with FORTH systems, and therefore no attempt is made to explain the facts.

Each word in the dictionary has a header which contains the following (in this order):

(a) two bytes for a link field to the previous entry in the dictionary.
(b) one byte for the number of characters in the name of the word. Bit 7 of this byte is set; bit 6 is the immediate flag and bit 5 the smudge bit.
(c) n bytes (maximum 31) for characters of the name: the last byte has bit 7 set.
(d) two bytes for the code pointer.
(e) then the parameter field, as long as necessary.

12 REVERSI

Unlike the producers of the vast majority of FORTH systems on the market, we believe it is essential to provide the user with an example of a well written, well documented FORTH program. If studied, it is far more instructive than the simple examples which are all that are usually given. With this in mind, we have supplied you with a copy of the well known game REVERSI, accompanied by a fully commented, well laid out listing of the the program, which is written entirely in FORTH. Indeed, the game could well have been sold in its own right. Please note that the version supplied on the microcartridge is an improvement on the documented version.

To load and run the game, type in:

```
LOAD_FILE MDV1_reversi_fth
```

Reversi, also known as Othello, is around 100 years old and is now a well established game, with regular world championships and regional championships.

The aim of the game is to end up with the most pieces on the 8x8 board. You and your opponent make moves alternately, using pieces which are black on one side and white on the other. The player who is black will always place them with black facing up, and the white player with white facing up.

To make a move, you must place a new piece such that you trap one or more of your opponent's pieces between the new piece and one or more of your own pieces, in a continuous (ie; no intervening vacant squares) straight line along a row, column or diagonal. You can only play on a vacant square - this is why the game cannot in any case last more than 64 moves excluding passes (you "pass" if you cannot make any move - it is then your opponent's turn). The move is completed by changing all the trapped pieces to your own colour (ie; by flipping them). If this sounds at all complex do not worry - SUPER REVERSI will not permit you to make an illegal move, so by actually playing you will soon pick up the game. Remember - a move must result in at least one flip!

The game is usually started with four pieces placed in the centre (as shown when you run the game), but SUPER REVERSI gives you the option of setting up your own starting position. Black always moves first - you are given the option at the beginning of the game to be either Black or White. Do not jump to the conclusion that the first player necessarily has an advantage - Reversi is a far more subtle game than that!

The game finishes when either all pieces have been played or when neither player can move. The player who has the most pieces showing on the board is then the winner (draws are hence possible) - SUPER REVERSI keeps track of the number of pieces for each side throughout the game. Note that it is only at the final position that the number of pieces of a particular colour decides the outcome - earlier on, it is not necessarily good strategy to maximise the number of pieces of your colour, for the simple reason that to do so would give your opponent more pieces to flip over at a later stage! Of course, you must have at least one piece on the board or else you will have to pass for the rest of the game (if you think for a minute, you will

If you have just begun playing SUPER REVERSI, here are some tips that should improve your playing strength:
(a) Do not 'grab' material - position is more important than material until the last stages of the game.
(b) In the beginning of the game, try to stay within the central 4x4 square area. The first player to move out of this area is often at a disadvantage.
(c) The most valuable squares are the corner squares as once occupied their occupier can (obviously) never be flipped. If the loss of a corner is inevitable then play should be directed towards blocking its effectiveness (eg; the corner A1 is much less useful for Black if Black also has A3 and White has A2).
(d) Edge squares other than corners are somewhat dangerous to occupy, especially those immediately adjacent to corner squares. They can provide an avenue of attack for your opponent culminating in his occupying a corner square.
(e) At every stage of the game try to make moves that, while not contradicting (a)-(d) above, reduce the number of options open to your opponent to a minimum.
(f) Long diagonals are useful only if a corner on that diagonal has been secured, or if the diagonals are for some other reason immune from attack.
(g) Squares immediately next to corners (eg; B2) are best left alone.
(h) Remember to count on your opponent playing well. Do not rely on the QL making oversights!

To interpret the final score, refer to the following table (which assumes that 64 pieces are on the board):

32-32	Drawn
33-31 to 35-29	Narrowly won
36-28 to 38-26	Comfortably won
39-25 to 41-23	Strongly won
42-22 to 49-15	A Smashing victory
50-14 or better	A Whitewash!

We wish you the very best of luck playing Digital Precision SUPER REVERSI - you are doing pretty well if you can beat it on level 5 and very well indeed if you can beat it at level 7 (the programmer - who received advice from Reversi experts in order to write the program - has himself yet to win, without cheating, on level 4 !!). In tests against other versions of REVERSI and OTHELLO for the Spectrum, QL, BBC micro and other computers (playing on equal time, at levels above those for beginners) Digital Precision SUPER REVERSI won every single time........

12.1 <u>GAME LISTING</u>

The complete listing of SUPER REVERSI follows. It is advisable to study the program carefully - it demonstrates SUPERFORTH in dynamic action!

```
          ( now we repeat the process for the letters A to H )
          ( along the bottom edge , very similar so no further )
          ( comments )

     65 430 243
     DO
          I 173 CURSOR
          DUP EMIT 1+ 25           ( letters 25 pixels apart )
     +LOOP DROP
               ( now we draw the horizontal lines, )
                         ( BLOCK_FILL is faster than LINE )
     173                 ( the bottom end of the line )
     33                  ( the top end of the line    )
     DO                  ( 33+9*17 > 173 so we loop 9 times )
          1              ( the line colour )
          202            ( the width, ie the line length )
          1              ( the height )
          237            ( the X start coordinate )
          I              ( the Y coordinate )
          BLOCK_FILL     ( draw the line )
          17             ( 17 pixels apart )
     +LOOP
               ( similarly draw the vertical lines )
     444 237
     DO
          1 2 136 I 33 BLOCK_FILL 25
     +LOOP 2 0 CSIZE  ;

HEX            ( convert to hexadecimal mode for now )

( FP converts a positive integer to floating point format, so )
( does not need the floating point package )

: FP DUP
     IF                  ( not zero )
          0              ( the provisional exponent )
          SWAP -10       ( decimal -16 )
          BEGIN
               OVER 4000 U<   ( repeat until top bit is a 1 )
          WHILE
               SWAP 2*        ( shift it 1 place left )
               SWAP 1-        ( decrement the exponent for )
                              ( each place shifted )
          REPEAT
          81F +               ( add the fiddle factor ! )
     ELSE                     ( integer is zero )
          0 0                 ( gives floating zero )
     THEN       ;

DECIMAL                       ( back to decimal )

( now lots of variables that are used )

VARIABLE C_COL           ( computer's colour )
VARIABLE P_COL           ( human player's colour )
VARIABLE COLOUR          ( temporary colour store )
VARIABLE P_SCORE         ( player's score )
VARIABLE C_SCORE         ( computer's score )
VARIABLE MEN             ( number of pieces on the board )
VARIABLE COMP            ( holds the computer's last move )
VARIABLE PLAYER          ( the player's last move )
```

```
HEX

CREATE SQU_VALUES   ( values in hex )
1010 , 1010 , 1010 , 1010 , 1010 , 2D19 , 211F , 1F21 ,
192D , 1019 , 111B , 1B1B , 1B11 , 1910 , 211B , 211F ,
1F21 , 1B21 , 101F , 1B1F , 0007 , 1F1B , 1F10 , 1F1B ,
1F07 , 001F , 1B1F , 1021 , 1B21 , 1F1F , 211B , 2110 ,
1911 , 1B1B , 1B1B , 1119 , 102D , 1921 , 1F1F , 2119 ,
2D10 , 1010 , 1010 , 1010 , 1010 , 1010 ,

DECIMAL

( now an array defining word which creates a board array )
( there will be one of these for each level of move looked )
( ahead by the QL, seven in all. there are 91 squares in )
( each board, numbered 0 to 90, the playing squares are )
( numbered 10 to 17, 19 to 26, etc. A one dimensional array )
( is used, rather than a two dimensional array, to avoid )
( multiplications, which are slow whatever the language used )
( to program the game, to index the board array only a simple )
( addition is needed )

: BD_ARRAY CREATE SIZE ALLOT       ( 92 bytes per board )
           DOES> ( OVER SIZE 1- U> IF )
                 ( ." Board array access error " QUIT THEN )
               +   ;

( the SUPERFORTH code commented out in BD_ARRAY checks  the
index )
( used when a board is accessed, it was very useful in )
( development of this program but is not needed in the final )
( version and so is removed to avoid slowing the game down )

( now define the 7 boards one for each depth of search )
( numbered P0 to P7, P for position )

BD_ARRAY P0_BOARD  BD_ARRAY P1_BOARD  BD_ARRAY P2_BOARD
BD_ARRAY P3_BOARD  BD_ARRAY P4_BOARD  BD_ARRAY P5_BOARD
BD_ARRAY P6_BOARD

VARIABLE BOARD_AD        ( used to indirectly execute one of )
                         ( above board arrays, so that common )
                         ( code can be used to access them )
                         ( Execution vectors can be used )

: BOARD BOARD_AD @ EXECUTE ;
( accesses the board whose code field address is loaded into )
( variable BOARD_AD , which will be loaded using the word ['] )

( SCORE calculates and displays the number of pieces belonging )
( to each player )

: SCORE
     0 P_SCORE !
     0 C_SCORE !         ( zero the scores )
     81 10               ( examine all the squares on the board)
     DO                  ( that can be occupied )
          I BOARD C@     ( get the square's value )
          DUP 16 <       ( occupied if less than 16 )
          IF
               P_COL @ =      ( is it the human's colour )
```

```
( three variables for every depth of search, ie for all seven )
( boards, described for P0, same for the rest )

VARIABLE P0_MOVES        ( points to P0's move list )
VARIABLE P0_SIZE         ( holds size of P0's move list )
VARIABLE P0_PTR          ( points to move being considered )
VARIABLE P1_MOVES    VARIABLE P1_SIZE    VARIABLE P1_PTR
VARIABLE P2_MOVES    VARIABLE P2_SIZE    VARIABLE P2_PTR
VARIABLE P3_MOVES    VARIABLE P3_SIZE    VARIABLE P3_PTR
VARIABLE P4_MOVES    VARIABLE P4_SIZE    VARIABLE P4_PTR
VARIABLE P5_MOVES    VARIABLE P5_SIZE    VARIABLE P5_PTR
VARIABLE P6_MOVES    VARIABLE P6_SIZE    VARIABLE P6_PTR

: INIT-BOARD                       ( initialises the screen )
                         ( called at the start of every game )
     #MAIN DRAW_SCR DRAW_SIDES     ( draws screen and board )
     ['] P0_BOARD BOARD_AD !       ( points to position 0 )
                                   ( so BOARD accesses that )
     SQU_VALUES 0 BOARD SIZE CMOVE ( copies initial square )
                                   ( values to position 0   )
     SQU_VALUES START_BOARD SIZE CMOVE  ( and to the board )
                                   ( holding the start position)
     0 MOVE_NO !                   ( zero move number )
     -1 QFLAG !                    ( clear the quit game flag )
     40 DRAW_PIECE 41 DRAW_PIECE   ( draw the four starting    )
     49 DRAW_PIECE 50 DRAW_PIECE   ( pieces )
     #TITLE
     0 90 10 0 0 BLOCK_FILL        ( draw the black and white )
     7 90 10 90 0 BLOCK_FILL ;     ( rectangles in #TITLE )

( EVALUATE calculates the value for a given move which is    )
( square value + w * men captured  )
( where w is 1 for moves 1 to 54 and 2 after that.            )
( It is called possibly a few times for a given move but the )
( square value is added in only once )
( The move value is later modified by subtracting the number )
( of moves the opponent can make )




: EVALUATE               ( n1 n2 n3 --- n1 n2 n3 )
                         ( n1 = square number of move )
                         ( n2 = step   see CHECK_8_ways )
                         ( n3 = square number of line end )
     MEN @ 11 / 3 - 1 MAX     ( factor w above )
     MEN_FLIPPED @ *          ( times men captured )
     NEW_MOVE @               ( TRUE if a new move )
     IF
                                   ( new move so )
          3 PICK BOARD C@ 16 - +   ( add in square value )
          0 NEW_MOVE !             ( clear new move flag )
          1 SIZE_PTR @ +!          ( increase move list size )
          3 PICK MOVE_AD @ @ C!    ( save the move )
     ELSE
          ."  I don't know"
     THEN
     250 SUSPEND_ME 0 ;         ( and stop for 5 seconds )

: COMP_COL C_COL @ COLOUR ! ; ( saves QL's colour in COLOUR )
: PLAY_COL P_COL @ COLOUR ! ; ( same for the player )


: INITPO ['] PO_BOARD BOARD_AD ! ;
( initialises the BOARD to the position 0 board )

EXVEC: OPERATION          ( used to flip pieces or make a move )

( this next word starts at a square and checks in one direction)
( to see if that square can be used for a move,if the square is)
( empty and next to a square of the opponent's colour then it )
( carries on until it finds it's own colour ie valid or an )
( square, which may be off the board, ie; invalid )

: CHECK_1_WAY        ( n1 n2 --- n1 n2 ) ( n1 = square number )
                     ( n2 = step value for the required )
                     ( direction, see CHECK_8_WAYS )
     DUP >R          ( save step on return stack )
     2DUP + BOARD C@      ( get value of adjacent square )
     DUP 16 <             ( if it is occupied ... )
     IF   COLOUR @ <>     ( ... with the opposite colour )
          IF   1 MEN_FLIPPED !      ( then set men captured )
               BEGIN                ( and carry on looking )
                    R@ +            ( move to next square )
                    2DUP +
                    BOARD C@        ( and access board )
                    DUP 16 <        ( if occupied ... )
                 IF COLOUR @ =      ( and our colour )
                    IF   R@ OPERATION  ( then evaluate or flip )
                    ELSE               ( else increment captured )
                       1 MEN_FLIPPED +! 0  ( and continue loop )
                    THEN
                 ELSE          ( not occupied so invalid move )
                    DROP -1    ( set flag to exit loop )
                 THEN
               UNTIL                ( end of loop )
          THEN DUP
THEN 2DROP R>  ;                    ( and tidy up stacks )
```

```
: YOUR_GO$  CLRMSG ."  Your move  ( eg H3 ENTER )" CR CR
                  ."  press 0 to list options"  ;

: MYMOVE       ( prints the QL's move if any in #MOVES )
     SCORE0 C@ DUP 0>           ( if QL has a valid move )
     IF 10 - 9 /MOD SWAP        ( convert to XY coords   )
        #MOVES C_TAB @ TAB
        65 + EMIT 49 + EMIT     ( and print them )
        C_TAB @ CR?             ( with a possible new line )
        10 2000 BEEP            ( and signal the move )
     ELSE
        DROP CLRMSG ."    I can't go"
        50 5000 BEEP            ( otherwise do this )
        0 OLD_SKILL !           ( don't know the best reply )
        200 SUSPEND_ME          ( and wait for 4 seconds )
     THEN ;


: FULL     ( --- flag ) ( TRUE if the board is full )
     P_SCORE @ C_SCORE @ + 64 = ;

: FLIP_PIECES         ( n1 n2 n3 --- n1 n2 n3 ) ( stack is as for)
                      ( EVALUATE, execution vector OPERATION  )
                      ( executes either of these two words to )
                      ( evaluate or make the move )
     OVER 3 PICK +    ( gives the end piece of the line )
     OVER 0< -        ( subtracts 1 if step is negative to avoid )
                      ( highlighting an existing piece )
     3 PICK           ( the start square )
     DO
        RED_PIECES @ 0=         ( if not drawing in red )
        IF COLOUR @             ( occupy the square with )
           I BOARD C!           ( the correct colour  )
        THEN
        I DRAW_MAN DUP          ( draw the piece and repeat the )
     +LOOP ;                    ( rest of the line )

: FLIP          ( ad --- ) ( makes the move held at ad  )
     ASSIGN OPERATION TO-DO FLIP_PIECES       ( ensures the move)
                                         ( is made and not evaluated )
     C@ ?DUP                             ( if a valid move is at ad )
     IF CHECK_8_WAYS  THEN ;             ( then make the move )

: MAKE_MOVE           ( ad --- ) ( makes a move on one of the )
                      ( boards P0 to P7 does not draw the board )
     ASSIGN DRAW_MAN TO-DO DROP     ( ensure no pieces are drawn)
     FLIP ;           ( and make the move at ad )

: DRAW_ALL_MEN        ( used when a move is retracted to redraw )
                      ( the whole board )
     #BOARD CLS DRAW_SIDES     ( clear and draw a blank board )
     80 10                     ( for every square )
        DO I 8 + I
           DO I BOARD C@ 16 <        ( which is occupied )
              IF I DRAW_PIECE THEN ( draw the piece )
           LOOP  9
        +LOOP
     SCORE ;          ( and print the new score )
```

```
: LEFT -25 L/R ;          ( moves 1 square left )
: RIGHT 25 L/R ;          ( moves 1 square right )

: U/D          ( n --- ) ( adjusts Y by n pixels up or down )
     Y @ +                ( get Y and add n )
     136 + 136 MOD        ( ensures rolls round top and bottom )
     Y ! SET_SQU ;        ( save and load square )

: UP  -17 U/D ;           ( moves 1 square up )
: DOWN 17 U/D ;           ( moves 1 square down )

: PUT_PIECE    ( n --- ) ( places a piece of colour n on the )
                          ( playing board PO )
     #MAIN SQUARE @ BOARD C!        ( store colour in the board )
     SQUARE @ DRAW_PIECE #BOARD ;   ( and draw it on the display)

: PUT_BLACK O PUT_PIECE ;      ( places a black piece )
: PUT_WHITE 7 PUT_PIECE ;      ( places a white piece )

: EMPTY                        ( blanks a square )
     4 PUT_PIECE    ( places and draws a green piece ie blank )
     SQU_VALUES SQUARE @ + C@       ( is the square a centre one)
     DUP 16 <             ( ie < 16 in SQU_VALUES, if so )
     IF DROP 37 THEN      ( then allocate a value to the playing)
                          ( board of 37 which is high )
     SQUARE @ BOARD C! ;  ( load board with the value )

: CLEAR              ( clears the whole board to the original )
                     ( starting position )
     SQU_VALUES O BOARD SIZE CMOVE       ( initialise the board )
     DRAW_ALL_MEN              ( draw all the men )
     #BOARD INIT_CSOR ;        ( and centre the cursor )

( the next prints the options available in #MESS )
: SET_HELP CLRMSG O 1 AT ."  Arrow keys move the cursor"
           CR ."  W or B places a white/black piece"
           CR ."  N      clears the square"
           CR ."  C      clears the board"
           CR ."  ESC    to terminate"    ;

: SET_POSITION            ( obeys the keys to set a position )
     #MOVES CLS           ( clear the moves window )
     INITPO               ( ensure setting board PO )
     INIT_CSOR            ( centre the cursor )
     SET_HELP             ( print the options )
     DRAW_ALL_MEN #BOARD ( draw all the men )
     BEGIN
        X @ Y @ CURSOR   ( position the cursor )
        CURSOR_ON KEY CURSOR_OFF DUP     ( get a key )
        CASE 192 OF    LEFT        ENDOF    ( left arrow )
             200 OF    RIGHT       ENDOF    ( right arrow )
             208 OF    UP          ENDOF    ( up arrow )
             216 OF    DOWN        ENDOF    ( down arrow )
              66 OF O PUT_PIECE ENDOF    ( B for black )
              87 OF 7 PUT_PIECE ENDOF    ( W for white )
              78 OF    EMPTY       ENDOF    ( N for none )
              67 OF    CLEAR       ENDOF    ( C )
              27 OF                ENDOF    ( ESC to exit )
            DEFAULT
        ENDCASE      27 =        ( repeat until ESC )
     UNTIL
```

```
: DRAW_RED           ( n --- ) ( draws the piece in red )
     DUP BOARD C@              ( get the colour of the piece )
     SWAP 2 OVER BOARD C!      ( and store a red piece )
     DUP DRAW_PIECE            ( draw it )
     BOARD C! ;           ( and restore the original colour )

: DRAW_MEN           ( ad --- ) ( draws the piece on square n  )
                     ( by first drawing it in red for 3 seconds )
                     ( and then in it's proper colour )
     DUP C@          ( get the move  if any ie not zero )
     IF ASSIGN DRAW_MAN TO-DO DRAW_RED  ( yes draw in red )
        -1 RED_PIECES !        ( set the red flag )
        DUP FLIP               ( and draw them )
        #MOVES 150 SUSPEND_ME  ( and wait 3 seconds )
     THEN
     ASSIGN DRAW_MAN TO-DO DRAW_PIECE   ( now draw the proper )
     0 RED_PIECES ! FLIP       ( colours and clear the flag )
     SCORE #MOVES  ;           ( print the new score )

: .MOVE                   ( n1 n2 --- ) ( prints a move )
     AT P0_PTR @ C@            ( get the move from P0's list )
     10 - 9 /MOD SWAP 65 +     ( and convert to ASCII ... )
     EMIT 49 + EMIT ;          ( ... and print it )

: BEST$                   ( prints the QL's best move so far )
     SKILL @ 2 >          ( only if playing level > 2 )
     IF #MESS 24 1 .MOVE                 ( print it )
     ."  (" SCORE0 2+ @ 4 .R ."  )  "    ( and it's value )
     THEN ;

: MOVE$                   ( prints the QL's move )
     SKILL @ 2 >          ( only if playing level > 2 )
     IF #MESS 24 3 .MOVE  THEN ;

: COMP-MOVE          ( generates a list of moves )
     ASSIGN OPERATION TO-DO EVALUATE     ( ensure evaluation )
     GEN_MOVES  ;                   ( and generate the moves )

( after this point there are a whole series of words which are )
( identical or very similar and which are numbered 0 to 6, )
( there is one word for each level of search or ply, )
( eg P3_SCORE calculates the value of a move at ply 3. It )
( would be more elegant to have written the program recursively)
( so that the same code could have been used but, would have )
( been much more difficult to understand, this is left as an )
( exercise for the future. Where these set of 6 or 7 identical )
( words occur, only the first is explained )

( the next 7 words generate a list of moves from each position )
( P0 to P6 )
: GENP0_MOVES
     INITP0                    ( ensure we use P0_BOARD )
     HEAP @ DUP           ( get the address of the work area )
     P0_MOVES ! P0_PTR !  ( and ensure the list of moves starts )
                          ( there )
     P0_PTR MOVE_AD !     ( point MOVE_AD to P0_pointer )
     0 P0_SIZE !          ( initialise the list size to zero )
     P0_SIZE SIZE_PTR !   ( point SIZE_PTR to P0_SIZE )
     0 0 P0_MOVES @ 2!    ( make first move 0 in case no moves )
     COMP-MOVE            ( and generate the list of moves )
     P0_PTR @ 2+ 2+ P1_MOVES !  ;  ( ensure the P1 move list )
```

```
( next we test move values to see if a higher value for a )
( move has been found, if so update the appropriate SCORE )

: TEST_P1_SCORE            ( n1 ---  ) ( n1 = new move value )
    SCORE1 @ SKILL @ 2 >       ( get SCORE1 and if skill > 2 )
    IF 24 2 .MOVE              ( we print the move and it's )
    26 TAB DUP ."  (" 4 .R ."  )"      ( value in brackets )
    THEN
    SCORE0 2+ @ 2DUP =         ( get QL's move value and if = )
                               ( new value we randomly select )
    IF 2DROP TIME DROP 1 AND 0 THEN >  ( one of these by )
                 ( using the QL's clock to choose )
    IF
       SCORE1 @ P0_PTR @ @ SCORE0 2!   ( update QL's best move)
       BEST$                           ( and print it )
       P_BEST? @ P_BEST !      ( update player's provisional )
    THEN ;                     ( best reply


: TEST_P2_SCORE                ( --- flag )
    SCORE2 @ DUP SCORE1 @ <         ( if SCORE2 < SCORE1 then )
    IF DUP SCORE1 !                 ( update SCORE1 )
       P1_PTR @ C@ P_BEST? !        ( and player's possible best)
    THEN
    SCORE0 2+ @ <  ;      ( flag is TRUE if SCORE2 < SCORE0 )
    ( ie; we can apply the alpha-beta algorithm: see below )

: TEST_P3_SCORE           ( --- flag )
    SCORE3 @ DUP          ( get P3 move value and save larger of)
    SCORE2 @ MAX SCORE2 !      ( this and P2 move value as P2 )
                               ( move value )
    SCORE1 @ >  ;         ( flag is TRUE if a better move found )

( the next 3 are uncommented:they are similar to TEST_P3_SCORE )
: TEST_P4_SCORE
    SCORE4 @ DUP SCORE3 @ MIN SCORE3 ! SCORE2 @ < ;

: TEST_P5_SCORE
    SCORE5 @ DUP SCORE4 @ MAX SCORE4 ! SCORE3 @ > ;

: TEST_P6_SCORE
    P5_SCORE + DUP SCORE5 @ MIN SCORE5 ! SCORE4 @ < ;
```

```
( now a set of words to initialise and generate a list of moves)
( from a given position. Again all very similar )

: GET_P0_MOVES            ( --- flag )
     32768 0 SCORE0 2!    ( initialise SCORE0 to the most )
                          ( negative integer and it's move to 0 )
     COMP_COL             ( ensure using the QL's colour )
     GENP0_MOVES          ( generate the list of moves )
     P0_SIZE @ DUP        ( get and duplicate the list size )
     P0_MOVES @ DUP       ( also the address of the move list )
     P0_PTR !             ( to initialise P0_PTR )
     SORT_HI              ( and sort the list in order )
     DUP 1 > SKILL @ 1 > AND  ;     ( flag is TRUE if list is )
                    ( longer than 1 move and SKILL > 1 ie need )
                    ( to go on to position P2 )

: GET_P1_MOVES
     COMP_COL P0->P1           ( set player's colour and copy )
     P0_PTR @ MAKE_MOVE        ( the P0 board to the P1 board )
                               ( and make the P0 move )
     PLAY_COL GENP1_MOVES P1_SIZE @     ( as above )
     P0_PTR SUB_SIZE           ( subtract no. of moves from the )
                               ( P0 value  etc as above )
     DUP P1_MOVES @ DUP C@ P_BEST? ! DUP
     P1_PTR ! SORT_HI 32767 SCORE1 ! SKILL @ 2 > ;

: GET_P2_MOVES
     PLAY_COL P1->P2 P1_PTR @ MAKE_MOVE
     COMP_COL GENP2_MOVES P2_SIZE @ P1_PTR SUB_SIZE
     DUP P2_MOVES @ DUP
     P2_PTR ! SORT_HI 32768 SCORE2 ! SKILL @ 3 > ;

: GET_P3_MOVES
     COMP_COL P2->P3 P2_PTR @ MAKE_MOVE
     PLAY_COL GENP3_MOVES P3_SIZE @ P2_PTR SUB_SIZE
     DUP P3_MOVES @ DUP
     P3_PTR ! SORT_HI 32767 SCORE3 ! SKILL @ 4 > ;

: GET_P4_MOVES
     PLAY_COL P3->P4 P3_PTR @ MAKE_MOVE
     COMP_COL GENP4_MOVES P4_SIZE @ P3_PTR SUB_SIZE
     DUP P4_MOVES @ DUP
     P4_PTR ! SORT_HI 32768 SCORE4 ! SKILL @ 6 > ;

: GET_P5_MOVES
     COMP_COL P4->P5 P4_PTR @ MAKE_MOVE
     PLAY_COL GENP5_MOVES P5_SIZE @ P4_PTR SUB_SIZE
     DUP P5_MOVES @ DUP
     P5_PTR ! SORT_HI 32767 SCORE5 ! SKILL @ 8 > ;

: BEST_P6_MOVE      ( this differs from the above  only in that)
               ( this is the last level and so we stop when we )
               ( have the highest valued move )
     PLAY_COL P5->P6 P5_PTR @ MAKE_MOVE
     COMP_COL GENP6_MOVES P6_SIZE @ P5_PTR SUB_SIZE
     P6_MOVES @ HIGH
     2+ @ >R 2DROP R>  ;

: PRUNE?  ( n1 --- n2 0 ) ( this decides if we prune the move )
          ( list or not pruning occurs on playing levels 5, 7  )
     1 MAX PRUNE @ MIN 0 ;   ( if PRUNE < n we keep it, lose n)
```

```
: BEST_MOVE                         ( gets the best QL's move )
                    ( and leaves it and it's value in SCORE0 )
   GET_P0_MOVES                     ( get the move list )
   IF 0                             ( if level > 1 go deeper )
     DO GET_P1_MOVES                ( to get the P1 move list )
       MOVE#              ( print P0 move under consideration )
       IF 1 MAX                     ( if level > 2 go deeper )
          PRUNE @ 2* MIN 0     ( prune to 10 moves if levels )
                               ( 5 or 7 )
          DO GET_P2_MOVES           ( get the P2 move list )
            IF PRUNE?          ( if level > 3 go deeper )
            DO GET_P3_MOVES    ( and get the P3 move list )
              IF PRUNE?        ( if level > 4 go deeper )
                 TRY_456_MOVES      ( carry on for plies 4,5,6 )
              ELSE DROP             ( level 4 )
                 P3_SCORE SCORE3 !  ( calculate the P3 move )
              THEN                  ( value and save it )
              TEST_P3_SCORE         ( is this better ? )
              IF LEAVE THEN         ( yes leave this loop )
              4 P2_PTR +!           ( else go on to next P3 move)
            LOOP
            ELSE DROP               ( level 3 )
               P2_SCORE SCORE2 !    ( calculate the P2 move )
            THEN                    ( value and save it )
            TEST_P2_SCORE           ( is this better ? )
            IF LEAVE THEN           ( yes leave this loop )
            4 P1_PTR +!             ( else go on to next P2 move)
          LOOP
       ELSE DROP                    ( level 2 )
          P1_SCORE SCORE1 !         ( calculate the P1 value )
          P1_MOVES @ C@        ( save it and if better save as )
          P_BEST? !            ( the player's best )
       THEN
       TEST_P1_SCORE           ( is this better ? )
       4 P0_PTR +!             ( move on to next move )
     LOOP
   ELSE DROP
   P0_PTR @ 2@ SCORE0 2!       ( level 1 top of list is best )
   THEN     ;


: GO                  ( generates and makes the QL's move )
     SKILL @ OLD_SKILL !       ( copy SKILL into OLD_SKILL )
     CLRMSG ." My move " SKILL @ 2 >    ( only if level > 2 )
     IF                                  ( do we print this lot )
       4 SPACES ." best so far" CR
       8 SPACES ." just considered" CR
       12 SPACES ." considering"
     THEN
     BEST_MOVE                 ( get the best QL's move )
     MYMOVE                    ( make the move  )
     ['] P0_BOARD BOARD_AD !   ( reset to P0_BOARD )
     COMP_COL                  ( set QL's colour )
     SCORE0 SAVE_MOVE          ( save the move in GAME )
     DRAW_MEN                  ( draw the new position )
     SCORE0 C@ COMP ! ;        ( and save the move in COMP )

: P_CR?             ( goes to a new line if player is white )
     #MOVES P_TAB @ CR? ;
```

```
: FIRST_MOVE          ( all the 4 possible first moves are )
                      ( equally strong so choose at random )
     COMP_COL                   ( set QL's colour )
     TIME DROP 3 AND            ( use clock to get a random )
                                ( number 0 to 3 )
     CASE                       ( which use to select a move )
          0 OF 32 ENDOF
          1 OF 42 ENDOF
          2 OF 48 ENDOF
          3 OF 58 ENDOF
          DEFAULT               ( can't occur
     ENDCASE
     SCOREO C! MYMOVE           ( store the move, make it  )
     SCOREO SAVE_MOVE           ( save it  and
     DRAW_MEN  ;                ( draw it )

: REVERSI                  ( executing this plays the game )
     BEGIN
          8 MODE                ( set 8 colour mode )
          INIT-BOARD            ( initialise the board )
          HEADER                ( the name of the game )
          M$BW                  ( choice of colours )
          #MOVES CLS            ( clear the moves window )
          SET_TABS              ( set tabs and print names )
          SKILL?                ( choose playing level )
          COLOUR @ WHITE =      ( if the player is white )
          IF FIRST_MOVE THEN    ( the QL goes first )
          BEGIN
               PLAY_COL         ( set player's colour )
               PLAYER-MOVE      ( and let him move )
               WIN              ( is the game over )
               IF -1
               ELSE             ( no, give the QL a go )
                  COMP_COL
                  GO
                  WIN           ( is the game over )
               THEN
          UNTIL                 ( repeat until it is )
          GAME_OVER             ( print result )
          AGAIN?                ( play again ?
     UNTIL
     FINISH  ;                  ( no, tidy up and finish )

HERE HEAP ! 1000 ALLOT     ( allocate 1000 bytes to working area )
( now open a big window and clear it, this gets rid of any )
( mess that may be outside the playing area, then close it )
0 OPEN SCR_512X256A0X0 2DUP #OUT 2@ 2SWAP #OUT 2!
5 PAPER CLS #OUT 2! CLOSE
END_FILE REVERSI           ( end of file and play the game )
                           ( note the order and they must be on )
                           ( the same line )
```

DIGITAL PRECISION

INDEX

DIGITAL PRECISION

```
: PLAYER_MOVE         ( allows the player to make his move )
    BEGIN
       PLAY_COL                                ( set player's colour )
       ASSIGN OPERATION TO-DO EVALUATE ( only evaluate when )
       SENPO_MOVES                             ( check if he can move )
       PO_MOVES @ C@                           ( if he can then )
       IF GET_MOVE DUP                         ( get his move )
         IF PLAYER C!                          ( if he has made a move)
            PO_MOVES @                    ( clear top of PO move list )
            0 OVER !
            PO_PTR                             ( initialise PO_PTR )
            -1 NEW_MOVE !                      ( ensure a new move )
            PLAYER C@                          ( get player's move )
            DUP BOARD C@ 15 >                  ( and check if he has )
              IF CHECK_8_WAYS                  ( made a valid move )
                 PO_MOVES @ C@ 0=
              THEN
              IF CLRMSG                        ( if not, tell him )
                 ." Illegal move, try again "
                 100 5000 BEEP                 ( with a rude noise )
                 150 SUSPEND_ME                ( wait 3 seconds )
                 #MOVES                        ( and clear his move )
                 P_TAB @ TAB 2 SPACES 0            ( from #MOVES )
              ELSE                             ( move is valid )
                 PLAYER SAVE_MOVE              ( so save it )
                 DRAW_MEN -1                   ( draw the new position)
                 P_CR?                         ( possibly a new line )
              THEN
         ELSE 1 THEN  ( player has swapped sides so leave )
      ELSE                                     ( he can't go so )
         0 PLAYER !                            ( clear PLAYER )
         CLRMSG ."    You can't go "           ( and tell him )
         50 5000 BEEP                          ( audibly )
         200 SUSPEND_ME -1                     ( wait 4 seconds )
         P_CR?                                 ( possibly a new line )
      THEN
    UNTIL ;           ( repet until valid move or can't go )

: WIN                 ( -- flag )
    COMP @ PLAYER @ + 0= FULL OR ;       ( flag is true to )
( indicate end of game, ie 64 pieces on the board or neither )
( side can move )

: AGAIN?              ( does he want another game )
    CLRMSG 6 2 AT ." Another game ? ( Y/N ) "
        CURSOR_ON KEY CURSOR_OFF
        89 = ;

: GAME_OVER           ( if he hasn't quit print the result in )
                      ( big letters )
    QFLAG @
    IF CLRMSG
         3 1 CSIZE 7 1 AT
         P_SCORE @ C_SCORE @ 2DUP =
         IF ." Game drawn" 2DROP
         ELSE > IF ."   You win" ELSE ."   QL wins" THEN
         THEN 2 0 CSIZE
         200 SUSPEND_ME
    THEN ;
```

```
: TRY_456_MOVES       ( works on ply 4,5 and 6 see BEST_MOVE )
   DO                          ( loop for all P3 moves )
     GET_P4_MOVES              ( get list of P4 moves  )
     IF                        ( if playing level > 6 go deeper )
       PRUNE?                  ( possibly prune no. of moves )
       DO GET_P5_MOVES         ( get list of P5 moves )
         IF 1 MAX 0            ( if level > 8 go deeper )
           DO BEST_P6_MOVE     ( and get best move at ply 6 )
              TEST_P6_SCORE    ( and test it and exit if )
              IF LEAVE THEN    ( we can leave this loop now )
              4 P5_PTR +!      ( or move on to next P5 move )
           LOOP
         ELSE DROP             ( levels 7,8 so calculate and )
           P5_SCORE SCORE5 !   ( load P5 value )
         THEN
         TEST_P5_SCORE         ( is the P5 move better ? )
         IF LEAVE THEN         ( yes we leave this loop now )
         4 P4_PTR +!           ( else we go on to next P4 move )
       LOOP
     ELSE DROP                 ( levels 5,6 so calculate and )
       P4_SCORE SCORE4 !       ( load P4 value )
     THEN
     TEST_P4_SCORE             ( is the P4 move better ? )
     IF LEAVE THEN             ( yes, leave this loop now )
     4 P3_PTR +!               ( else go on to next P5 move )
   LOOP ;


 ( now we have the word which finds the QL's best move, it )
 ( searches to a depth depending on the selected skill level )
 ( level 1 searches to a depth of 1 move   )
 (       2                        2 moves )
 (       3                        3  "    )
 (       4                        4  "    )
 (       5 and 6                  5  "    )
 (       7 and 8                  6  "    )
 (       9                        7  "    )
 ( levels 5 and 7 prune the number of moves examined to make )
 ( the QL move faster while still searching deeper )
 ( The alpha-beta algorithm is used to make the searches )
 ( faster, a detailed description is beyond the scope of this )
 ( description, see for example " Computer Gamesmanship " by )
 ( David Levy. Basically it terminates a search through a list )
 ( of moves if it finds a move that is better than one an )
 ( opponent can force, by selecting another move which has been )
 ( previously examined )
```

```
: HIGH    ( n1 ad --- n1 ad n2 ) ( finds the position n2 of the)
          ( highest valued move in the list at ad, n1 is the )
          ( number of moves in the list )
     DUP 2+ 2+            ( get address of next move )
     OVER 3 PICK          ( --- n1 ad1 ad2 ad1 n1 )
     DUP 1 >              ( only do if more than one move )
     IF 1
        DO >R R@ 2+ @              ( loop for n1-1 moves )
           OVER 2+ @ <             ( if the next < current )
           IF R> DROP DUP >R THEN  ( then save the next )
           2+ 2+ R>                ( and move to the next move )
        LOOP                       ( and repeat )
     ELSE DROP                     ( and tidy up stack )
     THEN SWAP DROP   ;

: SORT_HI      ( n ad ---   ) ( sorts a move list at ad with)
               ( n moves into order, highest at the top )
     OVER 1 >                 ( only if more than 1 move )
     IF OVER 1                ( loop n-1 times )
        DO HIGH               ( find position of highest )
           >R DUP 2@ R@ 2@    ( and swap with the top of )
           4 PICK 2! R> 2!    ( the list )
           SWAP 1- SWAP 2+ 2+ ( and repeat for the next move )
        LOOP                  ( in the list )
     THEN 2DROP ;

( the next few copy a position to another board for analysis )
( to another depth of search )

: P0->P1                      ( copies P0 to P1 )
     ['] P0_BOARD >BODY       ( get address of board P0 )
     ['] P1_BOARD             ( and code field address of P1 )
     DUP BOARD_AD !           ( ensure becomes current board )
     >BODY SIZE CMOVE ;       ( and copy P0 to P1 )

: P1->P2       ( see P0->P1 )
     ['] P1_BOARD >BODY ['] P2_BOARD DUP BOARD_AD !
     >BODY SIZE CMOVE ;

: P2->P3
     ['] P2_BOARD >BODY ['] P3_BOARD DUP BOARD_AD !
     >BODY SIZE CMOVE ;

: P3->P4
     ['] P3_BOARD >BODY ['] P4_BOARD DUP BOARD_AD !
     >BODY SIZE CMOVE ;

: P4->P5
     ['] P4_BOARD >BODY ['] P5_BOARD DUP BOARD_AD !
     >BODY SIZE CMOVE ;

: P5->P6
     ['] P5_BOARD >BODY ['] P6_BOARD DUP BOARD_AD !
     >BODY SIZE CMOVE ;

: SUB_SIZE     ( n ad --- n )
                     ( subtracts the size of a move list from )
                     ( the move value in the next higher move )
                     ( list, ie adjusts value for mobility )
     @ 2+                     ( get address of move value )
     OVER NEGATE SWAP +! ;    ( and subtract n from it )
```

```
                                         ( starts after the P0 list )
: GENP1_MOVES                ( see GENP0_MOVES )
     ['] P1_BOARD BOARD_AD !
     P1_MOVES @ P1_PTR ! P1_PTR MOVE_AD !
     0 P1_SIZE ! P1_SIZE SIZE_PTR ! 0 0 P1_MOVES @ 2!
     COMP-MOVE P1_PTR @ 2+ 2+ P2_MOVES !  ;

: GENP2_MOVES
     ['] P2_BOARD BOARD_AD !
     P2_MOVES @ P2_PTR ! P2_PTR MOVE_AD !
     0 P2_SIZE ! P2_SIZE SIZE_PTR ! 0 0 P2_MOVES @ 2!
     COMP-MOVE P2_PTR @ 2+ 2+ P3_MOVES !  ;

: GENP3_MOVES
     ['] P3_BOARD BOARD_AD !
     P3_MOVES @ P3_PTR ! P3_PTR MOVE_AD !
     0 P3_SIZE ! P3_SIZE SIZE_PTR ! 0 0 P3_MOVES @ 2!
     COMP-MOVE P3_PTR @ 2+ 2+ P4_MOVES ! ;

: GENP4_MOVES
     ['] P4_BOARD BOARD_AD !
     P4_MOVES @ P4_PTR ! P4_PTR MOVE_AD !
     0 P4_SIZE ! P4_SIZE SIZE_PTR ! 0 0 P4_MOVES @ 2!
     COMP-MOVE P4_PTR @ 2+ 2+ P5_MOVES ! ;

: GENP5_MOVES
     ['] P5_BOARD BOARD_AD !
     P5_MOVES @ P5_PTR ! P5_PTR MOVE_AD !
     0 P5_SIZE ! P5_SIZE SIZE_PTR ! 0 0 P5_MOVES @ 2!
     COMP-MOVE P5_PTR @ 2+ 2+ P6_MOVES ! ;

: GENP6_MOVES
     ['] P6_BOARD BOARD_AD ! P6_MOVES @
     P6_PTR ! P6_PTR MOVE_AD ! 0 P6_SIZE !
     P6_SIZE SIZE_PTR ! 0 0 P6_MOVES @ 2! COMP-MOVE ;

( now we work out move values by alternately subtracting and )
( adding values from the next higher move: what is good for the)
( opponent is bad for you so subtract his value )

: P1_SCORE            ( --- n ) ( calculate P1 move's value )
     P0_PTR @ 2+ @          ( get P0 move's value )
     P1_PTR @ 2+ @          ( and P1 move's value )
     - ;                    ( and subtract )

: P2_SCORE            ( --- n )
     P1_SCORE               ( get P1 move's value )
     P2_PTR @ 2+ @ + ;      ( and add P2 move's value )

( next 3 very similar )
: P3_SCORE P2_SCORE P3_PTR @ 2+ @ - ;
: P4_SCORE P3_SCORE P4_PTR @ 2+ @ + ;
: P5_SCORE P4_SCORE P5_PTR @ 2+ @ - ;
```

```
     SCORE                         ( print the new score )
     0 BOARD START_BOARD SIZE CMOVE          ( copy position to the )
     0 MOVE_NO !  ;                ( start and zero the move number )

: .OPTIONS            ( prints the options available on the )
                      ( player's turn )
     CLRMSG 0 0 AT
     ."  X exchange colours  ? best move" CR
     ."  S set up board      Q quit game" CR
     ."  R retract move      L skill level" CR
     ."  ESC return to SUPERFORTH"  CR
     ."  press any key to continue"
     KEY DROP   ;

: ?OPTION             ( n1 n2 --- flag ) ( selects an option )
     +                                  ( depending on n1+n2 )
     CASE 88 OF SWAP_SIDES ENDOF             ( X )
          82 OF RETRACT    ENDOF             ( R )
          81 OF QUIT_GAME  ENDOF             ( Q )
          63 OF .BEST      ENDOF             ( ? )
          76 OF SKILL? 0   ENDOF             ( L )
          83 OF SET_POSITION 0 ENDOF         ( S )
          79 OF .OPTIONS 0 ENDOF             ( O )
          27 OF ABANDON ENDOF                ( ESC )
          DEFAULT 0                          ( anything else )
     ENDCASE ;

: GET_KEY CURSOR_ON KEY CURSOR_OFF ;

: GET_MOVE            ( gets the player's move or option )
     BEGIN
        YOUR_GO$                        ( print message )
        #MOVES P_TAB @ DUP TAB          ( position cursor and )
        2 SPACES TAB                    ( blank any characters there)
        GET_KEY 65 - DUP 8 U< 0=        ( if the key is not A to H )
        IF 65 ?OPTION                   ( check for option )
        ELSE DUP 65 + EMIT              ( else print A to H )
          10 + GET_KEY                  ( get another key )
          49 - DUP 8 U< 0=              ( if not 1 to 8 )
          IF SWAP DROP 49 ?OPTION       ( check if option )
          ELSE DUP 49 + EMIT            ( else print 1 to 8 )
             9 * + GET_KEY 10 - ?DUP         ( look for ENTER )
             IF SWAP DROP 10 ?OPTION         ( if not check option )
             ELSE -1                    ( if none of these ensure )
             THEN                       ( repeat the loop until a )
          THEN                          ( valid move is read )
       THEN
     UNTIL  ;

: M$BW           ( requests the colours )
     CLRMSG ." Do you wish to play " CR
     ." black or white ? ( B or W ) "
     WHITE BLACK
     CURSOR_ON KEY CURSOR_OFF           ( get a key )
     CLRMSG 66 =                        ( if a W then set the )
     IF SWAP THEN                       ( colours appropriately )
     OVER COLOUR ! C_COL ! P_COL ! ;
```

```
: SWAP_SIDES          ( called to swap sides ie cheat ! )
     P_COL @ C_COL @ P_COL ! C_COL !    ( swap the colours )
     1 PLAYER !                    ( to ensure the QL plays )
     P_TAB @ C_TAB @ P_TAB ! C_TAB !    ( swaps the tabs for )
                                         ( #MOVES )
     SET_TABS                      ( and swaps the titles )
     0 -1 ;           ( flags to ensure we leave GET_MOVE )

: RETRACT             ( the option to takee back a move )
     START_BOARD 0 PO_BOARD SIZE CMOVE   ( copy the starting )
                                         ( board to PO )
     INITPO                ( select PO board )
     0 OLD_SKILL !         ( the QL won't know the best reply )
     -1 MOVE_NO +!         ( go back one move )
     MOVE_NO @ 0<          ( if it is now negative, we are at )
                           ( starting position so )
     IF 0 MOVE_NO !        ( clear move number )
        100 5000 BEEP      ( make a rude noise )
        CLRMSG             ( output a message )
        5 2 AT ." At starting position"
        150 SUSPEND_ME     ( for 3 seconds )
     ELSE MOVE_NO @ 0>     ( otherwise if at least one move on )
        IF MOVE_NO @ 0                   ( for every move made )
           DO I GAME C@ DUP 127 >        ( make the move in GAME)
              IF WHITE ELSE BLACK THEN   ( play the correct   )
              COLOUR !                   ( colour, save the move)
              127 AND PLAYER C!          ( in PLAYER )
              PLAYER MAKE_MOVE     ( and make the move without )
           LOOP                    ( drawing it )
        THEN
        SWAP_SIDES 2DROP           ( swap colours )
        DRAW_ALL_MEN          ( and draw the new board )
        #MOVES CLS            ( clear the moves )
     THEN
     PLAY_COL 0               ( the flags to exit GET_MOVE )
     ASSIGN OPERATION TO-DO EVALUATE ;   ( reassign OPERATION )

: QUIT_GAME 0 QFLAG ! 0 PLAYER ! 0 COMP ! 0 -1 ;
( leaves the current game and requests another )

( variables for setting up a new board )
VARIABLE SQUARE       ( holds the square number of the cursor )
VARIABLE X            ( the X and Y coordinates of the cursor )
VARIABLE Y            ( in #BOARD )

: INIT_CSOR           ( initialises the cursor and square )
     40 SQUARE ! 83 X ! 73 Y ! ;

: SET_SQU             ( converts the XY pixel coordinates to )
                      ( a square number and loads SQUARE       )
     X @ 25 /         ( each square 25 pixels wide )
     73 +             ( fiddle factor, Y 0 at top, square 1 )
                      ( at bottom )
     Y @ 17 /         ( each square 17 pixels high )
     9 * -            ( times the squares per row and subtract )
     SQUARE ! ;       ( to leave and store the square number )

: L/R           ( n --- ) ( adjusts X by n pixels left or right )
     X @ +                ( get X and add n )
     200 + 200 MOD        ( ensures rolls round left and right )
     X ! SET_SQU ;        ( and save and load SQUARE )
```

```
: CHECK_2_WAYS        ( n1 n2 --- n1 -n2 ) ( checks whether a )
                      ( move on square n1 is valid in direction)
                      ( n2 and direction -n2 )
    CHECK_1_WAY                 ( checks in one direction eg NW )
    NEGATE CHECK_1_WAY ;        ( and the opposite way eg SE   )


( The next word uses the linear square numbering system to )
( check in 8 directions, to move north, say, a step of 9 is )
( needed eg square 10 + 9 gives square 19, adding another 9 )
( gives 28 etc, this moves north on the board, the directions )
( are defined by steps )
( 9  north            -9   south      )
( 1  east            .-1   west       )
( 10 north-east       -10  south-west )
( 8  north-west       -8   south-east )

: CHECK_8_WAYS        ( n --- ) ( checks whether a move on square)
                      ( is valid in all 8 directions. )
    1 CHECK_2_WAYS              ( checks east and west )
    8 - CHECK_2_WAYS            ( then south and north )
    1+ CHECK_2_WAYS             ( then north-east and south-west )
    2+ CHECK_2_WAYS             ( and  south-east and north-west )
    2DROP  ;                    ( tidy up stack )

: GEN_MOVES                ( --- ) ( generates the list of valid )
                           ( moves in a given position by        )
                           ( testing every square on the board   )
    80 10                  ( outer and inner loops used to avoid )
    DO I 8 + I             ( testing off-board squares eg 18     )
       DO -1 NEW_MOVE !         ( set new move flag )
          I BOARD C@ 16 >       ( if square is empty ... )
          IF I CHECK_8_WAYS     ( then check in 8 directions to  )
          THEN                  ( see if it is a valid move )
       LOOP   9                 ( and repeat for the rest the row)
    +LOOP ;                     ( and the rest of the rows )

: FINISH              ( to return to SUPERFORTH, tidies the screen
)
    #MESS 5 1 BORDER CLS        ( clear #MESS and the border   )
    #MAIN 0 20 AT ;             ( and position the cursor      )

: ABANDON  FINISH ." Game abandoned " ABORT  ;
( the abandon game message )

: CR?                 ( decides whether to go to a new line    )
                      ( in the moves window )
    10 =              ( new line if the tab is 10 )
    IF CR SPACE THEN ;  ( SPACE forces the QL to new line )

: SKILL?         ( requests the level of skill desired by the )
                 ( player and sets the tree pruning flag       )
    CLRMSG ." Level of skill ( 1 to 9 )" CR 0
    BEGIN DROP                       ( drop an invalid key )
         CURSOR_ON KEY CURSOR_OFF    ( get a key )
         49 - DUP 9 U<               ( until in the range 1 to 9 )
    UNTIL
    1+ SKILL ! SET_TABS              ( save in SKILL and display )
    SKILL @ DUP 5 = SWAP 7 = OR      ( in #TITLE, if 5 or 7 )
    IF 5 ELSE 100 THEN PRUNE ! ;     ( then set the prune flag )
```

```
          4 MOVE_AD @ +!         ( move on the move list )
                                 ( pointer, 4 bytes per move )
          MOVE_AD @ @ 2- !       ( save the move's value )
     ELSE
          MOVE_AD @ @ 2- +!      ( move already in list so )
                                 ( on extra value )
     THEN  ;

: HEADER                   ( writes the name at the screen top )
     #MAIN 3 1 CSIZE             ( in large charactars )
     6 120 24 158 2 BLOCK_FILL   ( on yellow background )
     162 4 CURSOR                ( at the top centre )
     6 STRIP ." REVERSI"         ( print it )
     5 STRIP 2 0 CSIZE ;         ( and restore strip and size)

: OFF 2+ 2@ CLOSE ;        ( ad --- ) ( closes channel whose ID )
                           ( is at address +2 as saved by #CON )

: CLOSE_ALL                ( closes all display windows )
          ['] #TITLE OFF
          ['] #MOVES OFF
          ['] #SCORE OFF
          ['] #MESS  OFF
          ['] #BOARD OFF  ;

: SAVE_MOVE           ( ad --- ad ) ( saves the move being made )
                      ( in array GAME, + 128 if a white move )
                      ( ad points to the move )
     DUP C@                ( get the move )
     COLOUR @              ( if the colour is white )
     IF 128 + THEN         ( then add 128 )
     MOVE_NO @ GAME C!     ( save move in GAME )
     1 MOVE_NO +! ;        ( and increment the move number )

: SET_TABS            ( sets the tabs for #TITLE and prints the )
                      ( column headings )
     2 10                  ( columns 2 and 10 )
     C_COL @ BLACK =       ( if QL's colour is black )
     IF SWAP THEN          ( then reverse the tabs )
     C_TAB ! P_TAB !       ( and save in variables )
     #TITLE 0 25 CURSOR    ( position cursor in #TITLE )
     SKILL @               ( get playing level )
     C_COL @ BLACK =       ( if QL is black )
     IF ."  QL/" .         ( print QL first )
           ."   HUMAN"     ( then the other )
     ELSE
          ."  HUMAN   QL/" . ( else the other way round )
     THEN ;

: CLRMSG #MESS 1 1 BORDER CLS 0 1 AT ;
( clears the message window and sets the cursor position )

: .BEST               ( prints the player's best move as held in )
                      ( variable P_BEST )
     CLRMSG OLD_SKILL @ 1 >     ( OLD_SKILL > 1 if the QL knows )
     IF                         ( so it prints it out )
          P_BEST @ ."  Your best move is "
     10 - 9 /MOD SWAP           ( converts to XY grid reference )
     65 + EMIT 49 + EMIT        ( and prints them )
```

```
               IF
                    1 P_SCORE ( yes, prepare to add to the )
                              ( player's score )
               ELSE           ( otherwise the QL's score )
                    1 C_SCORE
               THEN
               +!             ( and add it )
          ELSE                ( square is empty so do nothing )
               DROP
          THEN
     LOOP
     #SCORE                            ( output to #SCORE )
     P_SCORE @ C_SCORE @ 2DUP + MEN !  ( save the total number)
                                       ( of men )
     C_COL @ WHITE =     ( swap the scores if the QL is white )
     IF SWAP THEN
     24 14 CURSOR .      ( and print the two scores at the )
     120 14 CURSOR . ;   ( the correct position )


: PUT_COORD         ( used by DRAW_PIECE to copy the circle )
                    ( parameters to PARAMETERS for CIRCLE )
     PARAMETERS + 6 CMOVE  ;

( the next draws a coloured circle on the board on the screen )

: DRAW_PIECE   ( n --- ) ( where n is the square number )
     #MAIN               ( draw on #MAIN )
     DUP 10 - 9 /MOD     ( converts the square number to an )
                         ( XY reference to access Y_CEN and )
                         ( X_CEN to draw the circle )
     Y_CEN 18 PUT_COORD  ( copy the Y centre coordinate )
     X_CEN 24 PUT_COORD  ( and the X centre )
     BOARD C@            ( get the squares value )
     INK                 ( to set the ink colour )
     PARAMETERS          ( the address of the parameters )
     FILL_ON CIRCLE FILL_OFF  ( draw a disc )
     1 INK    ;          ( and restore ink colour )

( now an execution vector that is used to execute DRAW_PIECE )
( or not, it is used so that we can later use some words that )
( may or may not need to actually draw a piece )

EXVEC: DRAW_MAN

( more variables )

VARIABLE HEAP                           ( base of work area )
( six to hold the move values of different level positions )
2VARIABLE SCORE0    VARIABLE SCORE1     VARIABLE SCORE2
VARIABLE SCORE3     VARIABLE SCORE4     VARIABLE SCORE5
VARIABLE MOVE_AD         ( points to a position's move list )
VARIABLE SIZE_PTR        ( points to a move list's size )
VARIABLE SKILL 2 SKILL ! ( the playing level )
VARIABLE C_TAB           ( QL's tab value for #MOVES )
VARIABLE P_TAB           ( player's tab value )
VARIABLE P_BEST?         ( provisional player's best move )
VARIABLE P_BEST          ( best move )
VARIABLE OLD_SKILL       ( previous playing level )
VARIABLE RED_PIECES      ( defines temporary red pieces )
VARIABLE MEN_FLIPPED       ( the number of pieces flipped over )
                           ( in one direction for a move )
VARIABLE NEW_MOVE          ( a flag to indicate a new move )
VARIABLE QFLAG             ( a flag to indicate Quit game )
VARIABLE MOVE_NO           ( the move number )
VARIABLE PRUNE             ( indicates whether the list of moves )
                           ( is pruned or not )

7 CONSTANT WHITE           ( the value of white used by the QL  )
0 CONSTANT BLACK           ( the value of black )
92 CONSTANT SIZE           ( the number of bytes needed to hold )
                           ( a game position )

: ARRAY CREATE ALLOT DOES> + ;
( creates a byte array which when executed adds the index to )
( the array start address, the size of the array is on the )
( stack )

64 ARRAY GAME              ( used to store the moves made )

CREATE START_BOARD 92 ALLOT   ( used to hold the starting )
                              ( position of a game )

: FPARRAY CREATE DOES> SWAP 6 * + ;
( used to create an array of floating point numbers, each of )
( which is 6 bytes long )

: FP, FP , , ,  ;          ( compiles a floating point number )

FPARRAY X_CEN         ( an array of the x coordinates of the )
                      ( centres of the playing squares, used to )
                      ( draw the pieces )
          185 FP, 203 FP, 222 FP, 241 FP,    ( pixel numbers )
          260 FP, 279 FP, 297 FP, 315 FP,

( similarly for the Y coordinates )

FPARRAY Y_CEN  82 FP, 100 FP, 117 FP, 134 FP,
              152 FP, 169 FP, 185 FP, 202 FP,

CREATE PARAMETERS 0 FP, 6 FP, 1 FP, 0 FP, 0 FP,
( the parameter list for CIRCLE : see the SUPERFORTH manual )

( now use a temporary area of RAM to set the SCALE of the )
( display, the next words are executed not compiled )

HERE 500 + 18 0 FILL            ( clear it )

244 FP HERE 512 + 2!            ( the scale factor in FP format )
DROP  HERE 500 + SCALE          ( and set the scale )

( next we allocate to every square of the board a value which )
( indicates how good it is, the squares value is 16 less than )
( the value stored. When a game is played a value of less than )
( 16 indicates it is occupied and gives it's colour, a value )
( of more than 16 means unoccupied, equal to 16 is off the )
( board there are 92 squares, 64 to play on, and 28 round 3 )
( of the edges, this makes off-board detection easy. )
( These values are compiled into an array which is copied )
( to the starting position of every game. )
( REVERSI version 1.2   copyright 1985 G.W.Jackson )

CLS 3 1 CSIZE CR .(       LOADING REVERSI) CR 1 0 CSIZE

( #CON is a defining word that creates and opens a )
( display window and saves the channel ID. When the )
( newly created word is executed it makes that window )
( the input/output window by loading it's channel ID  )
( into #IN and #OUT )

: #CON 2CONSTANT          ( creates a double length constant )
     DOES> 2@ 2DUP        ( leaves the channel as TOS, twice )
          #IN 2!          ( which it loads into #IN )
          #OUT 2! ;       ( and #OUT )

( now we open all the windows used in the game )

0 OPEN SCR_180X50A52X44   #CON #TITLE   ( the red title window )
0 OPEN CON_180X80A52X93   #CON #MOVES   ( for the  moves )
0 OPEN SCR_180X26A52X172  #CON #SCORE   ( for the score  )
0 OPEN CON_420X54A52X199  #CON #MESS    ( for messages and  )
                                        ( information )
0 OPEN CON_202X136A268X44 #CON #BOARD   ( the playing board )
#OUT 2@ #CON #MAIN            ( covering the whole screen    )

: INIT_SCR DUP PAPER STRIP INK DUP BORDER ;
( initialise screens, used by the below words to set the )
( colour and border of each display window )

: DRAW_SCR #MAIN          ( makes #MAIN the current window )
         0                ( the border width )
         1                ( the ink colour      )
         5                ( the paper and strip colour )
         INIT_SCR         ( set the above parameters     )
         CLS              ( and clear the window )
                          ( now do the same for the rest of the )
                          ( windows )
        #TITLE 0 7 2 INIT_SCR CLS
        #MOVES 0 1 6 INIT_SCR CLS
        #SCORE 0 7 1 INIT_SCR CLS
        56 2 CURSOR ." SCORE"           ( print the heading )
        #MESS  1 1 5 INIT_SCR CLS
        #BOARD 0 1 4 INIT_SCR CLS       ;

( DRAW_SIDES draws the grid of the playing board and prints )
( the square coordinates round the sides )

: DRAW_SIDES  #MAIN       ( done on #MAIN because the letters & )
                          ( numbers are outside #BOARD )
     56                   ( the ASCII code for 8 )
     173                  ( the end pixel Y coordinate )
     39                   ( the start pixel Y )
     DO                   ( loop to print numbers 8 to 1 )
         222 I CURSOR     ( position the pixel cursor )
         DUP EMIT         ( print the digit )
         1-               ( decrement the ASCII code )
         17               ( the loop step, digits are 17 pixels )
                          ( apart vertically )
     +LOOP                ( and repeat for the next digit )
     DROP                 ( drop the TOS, no longer needed )
```

realise why!).

You make a move by typing the grid reference of the move: for example, H3, followed by ENTER. Any other key either selects one of the command options listed below or cancels the move. Note that commands are accepted in upper case only — so if you are leaving the computer unattended in the middle of an important game, press CAPS LOCK to ensure the position and game are not tampered with! Press CAPS LOCK again to re-enable command entry.

O	....	Display Options
X	....	Exchange sides (ie; cheat!)
S	....	Setup a new position
R	....	Retract one or more moves (ie; cheat!)
?	....	Hint - Suggest a move (ie; cheat!)
Q	....	Quit the game - ie; Resign
L	....	Change skill level
ESC	....	Return to SUPERFORTH
CTRL+C	....	Return to SuperBASIC

Note that the O option makes the above table redundant - you need not have the manual open to play Reversi.

There are 9 playing levels ranging from the easiest at level 1 to the hardest at level 9 (in which the QL has a 7 move look-ahead !). On levels 3 and above, while the QL is thinking, it displays the move currently being examined, the best move so far and the last move considered: with these last two it also displays a value which indicates how good the move is (the higher the value the better for the QL). This makes waiting for the QL to move very interesting even when it is set to long playing times!

The levels (and the approximate time taken) are:

1	....	Beginner	0.1 seconds
2	....	Novice	2 seconds
3	....	Intermediate	30 seconds
4	....	Fairly strong	1 minute
5	....	Strong	2.5 minutes
6	....	Very Strong	5 minutes
7	....	Master	10 minutes
8	....	Expert	30 minutes
9	....	Champion	1.5 hours

When a move is made, the pieces affected are displayed in red for a few seconds, so that you can the move's effect.

If you want to see the computer play against itself, press X repeatedly.

Setup mode (S) is useful either to solve Reversi problems or to return to a position that had to be abandoned. Any position may be setup but some may not be much fun (ie; the empty board, which will of course result in a drawn game!). The keys to be used for setup are displayed on the setup screen, but here is a list of them anyway:

Arrow Keys	....	Move the cursor
W	....	Put a White piece on the square
B	....	Put a Black piece on the square
C	....	Clear the board
N	....	Clear the square
Esc	....	Terminate setup mode

The system uses indirect threaded code: ie; each call to a secondary points to the code pointer of that secondary, which itself points to the code to be executed for that word.

11.4 INFORMATION FOR MACHINE CODE USERS

DP SUPERFORTH uses the following registers, which, if used by some machine code, must be saved before and restored after the machine code:

A0.L holds the SUPERFORTH address (ie; 16 bit relative to A2) of the USER variables.

A1.L is the IP, or interpretive pointer, that points to the parameter field of the SUPERFORTH word currently being executed. A1.L is pushed onto the return stack when a secondary is called.

A2.L holds the absolute address of location 0 in the SUPERFORTH dictionary: all SUPERFORTH addresses are relative to A2.L

A3.L is the data stack pointer. It holds an absolute address and points to the second item on the stack.

A4.L is the return stack pointer, an absolute address.

A7.L is used as an internal stack pointer to temporarily hold data during QDOS calls. It is also used by QDOS.

D2.W is the top of the data stack.

In addition to these, the other registers are used for various operations and cannot be guaranteed to remain uncorrupted, but changing them in a machine coded definition will not matter.

If a machine code word is inserted, it must end with the following code (in HEX) or a branch to such a sequence, which is the well known NEXT sequence:

```
HEX      3219      MOVE.W (A1)+,D1
         3A72      MOVE.W 0(A2,D1.W),A5
         1000
         4EF2      JMP     0(A2,A5.W)
         D000
```

11.5 ABSOLUTE RAM ADDRESSES

The absolute address of the dictionary is held as a double variable in SUPERFORTH location 32776: ie; typing

```
32776 2@
```

will leave the absolute address of SUPERFORTH location 32768 as a double-number on the stack. This may vary when the SUPERFORTH system is loaded, depending on what other tasks are running before SUPERFORTH is loaded and whether extended RAM is fitted to the QL.

10.3 SOUND GENERATION

Some SUPERFORTH words are provided to facilitate use of the QL's sound generator; these include simple beeps and a defining word.

BEEP (n1 n2 ---) generates a single tone: n1 is the pitch (in the range 0 to 255) and n2 the duration (in units of 72 microsecs). If n2 is zero, the sound will continue indefinitely until another BEEP or SILENCE .
Eg; 50 5000 BEEP

BEEPING (--- flag) tests the sound generator and leaves the flag TRUE if sound is being generated, otherwise FALSE.

SILENCE (---) silences the sound generator.

SOUND (n1 n2 n3 n4 n5 n6 n7 n8 ---) is a defining word used in the form

SOUND <name>

to enter a word called <name> in the dictionary which, when executed, will generate the sound defined by parameters n1 to n8, which are (see QL User Guide):

n1 fuzziness	range 0 to 15
n2 randomness	range 0 to 15
n3 wrap	range 0 to 15
n4 step grad_y	range -8 to 7
n5 duration	range 0 to 65535
n6 interval grad_x	range 0 to 65535
n7 pitch 2	range 0 to 255
n8 pitch 1	range 0 to 255

Eg; 0 0 15 1 1500 100 50 1 SOUND ZAP
Now type ZAP to generate the sound (this sound is already in the dictionary).

10.4 TIME AND DATE

Words are included to enable you to set and read the internal clock of the QL. All times are expressed in seconds and affect the time and date.

ADJUST_TIME (d ---) adds double-number d to the time. d is in seconds and may be negative, eg;
100. ADJUST_TIME adds 100 seconds

DATE$ (--- ad) leaves the address of a string representing the date and time on top of the stack. The string is stored in the standard SUPERFORTH format, ie; the first byte is the number of characters:
DATE$ COUNT TYPE prints the date and time
DATE$ 12 + 9 TYPE prints the time only

DAY$ (--- ad): as DATE$, except that the string is the day of the week, eg;
DAY$ COUNT TYPE

RUNS (---) must only be used after JOB (see above)

OWN_USERS (--- ad)
OWN_PAD (ad --- ad)
OWN_TIB (ad --- ad)
OWN_BUF (ad --- ad)

These four words reserve dictionary space for USER variables, PAD, TIB and a block buffer respectively. ad in all cases is the address of the USER variable area. OWN_USERS must be used immediately before the other three, which are optional: eg; a sequence might be:

```
OWN_USERS OWN_PAD 8 16 1 JOB FRED RUNS MARY
```

If a task inputs data or outputs data, it must use its own USER variables and PAD (for output) and TIB (for input). An input buffer must be used if data is to be read from mass storage by the task.

An additional requirement for tasks using WORD and the graphics words ARC, CIRCLE, LINE, POINT and SCALE is an area of dictionary for working (for an arithmetic stack for QDOS). To allocate this, add:

```
310 ALLOT
```

after the task is created using JOB ... RUNS ...

10.2.3 Task activation

ACTIVATE (d1 d2 n ---) is used to start a task with job identity d2. d1 = 0 for the current job to continue and -1 to suspend the current job until the activated job is finished (do not use d1 = -1 with CLOCK because CLOCK never terminates). n is the new task's priority: 1 is the highest priority and 127 the lowest priority.

Eg; 0 0 ?JOB_ID CLOCK 15 ACTIVATE
starts the clock.

EXEC (---) is used to activate a machine code task from mass storage, just like SuperBASIC EXEC. The new task's identity is left in variable JOB_ID .

Eg; EXEC MDV1_TASK (assuming a task named TASK is held on MDV1_).

START (d n1 ---) is used as START <name> to start an inactive job with priority n1. If d is 0, the current job continues; if d is -1, the current job is suspended indefinitely.

Eg; to start the clock with priority 10 (assuming the clock has never been activated):

```
0 0 10 START CLOCK
```

10.2.4 Suspending and restarting tasks

FREEZE (d n ---) suspends a task with identity d for n fiftieths of a second, eg;

```
?JOB_ID CLOCK 500 FREEZE
```

TIMEOUT (--- n): a constant defining the timeout of an input or output operation, it is initially -1, which means that input and output operations will wait indefinitely if the input or output device is not ready or has no data. If TIMEOUT is positive, it defines the length of time the QL will wait for input or output in fiftieths of a second. This may be used, for example, to read the keyboard but not wait if no key has been pressed. Always be careful to restore it to -1 afterwards.

10.1.1 Redirection of input/output

This may be achieved using the above words in the following way, for example to output to a new screen window:

```
2VARIABLE #MESSAGES
0 OPEN SCR_420X44A52X209 #MESSAGES 2!
```

and whenever you want to output to this window you use the sequence (of course, you can define a word to do this):

```
#MESSAGES 2@ #OUT 2!
```

any output now goes to this new window. To revert to the original, type:

```
#DEFAULT #OUT 2!
```

A similar sequence is used to redirect input.

10.1.2 Printer operation

Certain words are already provided which perform the redirection, enabling you to output to the printer:

#PRINT (--- ad): a double variable used to hold the channel ID for the printer.

PRINTER_IS (---) defines the characteristics of your printer (see the QL User Guide for details), opens a channel to the printer and saves the channel ID in double variable #PRINT: eg;

```
PRINTER_IS SER1E
```

PRINTER_ON (---) simply selects the printer as the output device by loading #PRINT into #OUT. It also ensures that the prompt ok is output to the display and not the printer, and that CLS does not send nasty characters to the printer.

PRINTER_OFF (---) restores the default output device to #OUT.

PRINTER_CLOSE (---) closes the printer channel.

```
COS    ( fp1 --- fp2 )
SIN    ( fp1 --- fp2 )
TAN    ( fp1 --- fp2 )       the usual trigonometric
COT    ( fp1 --- fp2 )       functions; angles must be
ARCSIN ( fp1 --- fp2 )       expressed in radians.
ARCCOS ( fp1 --- fp2 )
ARCTAN ( fp1 --- fp2 )
ARCCOT ( fp1 --- fp2 )

SQRT ( fp1 --- fp2 )        the square root
LN   ( fp1 --- fp2 )        the natural logarithm
LOG10 ( fp1 --- fp2 )       log to the base 10
EXP  ( fp1 --- fp2 )        e to the power fp1
^  ( fp1 fp2 --- fp3 )      fp1 to the power fp2
```

Conversions between floating point numbers and integers are achieved by:

```
F->S   ( fp --- n )  floating to nearest single integer
F->D   ( fp --- d )  floating to nearest double integer
INT    ( fp --- n )  truncate fp to single integer
S->F   ( n --- fp )  single integer to floating
D->F   ( d --- fp )  double integer to floating
```

Input and output of floating point numbers is achieved with:

```
F.     ( fp --- )    which prints a floating point number on
                     the display
F$     ( --- fp )    which converts the next word into a
                     floating point number, eg;

                     F$ 3.14159 FCONSTANT PI        or
                     F$ 123.45E83
```

Use of all these words is straightforward. Those words with integer equivalents are used in the same way. Others, such as the trigonometric functions, are used as in the following example:

assuming PI defined as above,

```
          PI 2 S->F F/ SIN F.  ( to print sin(pi/2) )
```

dictionary. This time, however, the name being searched for is held in memory at ad1 as a counted string. If the name is found, ad2 is the compilation address of the name and n has one of two values: if the word found is immediate, then n is set to 1; if not immediate, then n is set to -1. If the name is not found, then ad2 = ad1 and n is set to 0: eg;

```
: LOCATE 32 WORD FIND . U. ;            and try
LOCATE DUP          ( displays -1 and an address )
LOCATE IF           ( displays  1 and an address )
LOCATE xyz          ( displays  0 and an address )
```

ID. (ad ---) displays the name of the dictionary entry whose header starts at ad, often used in conjunction with LATEST.

LATEST (--- ad) puts the address of the last word defined in the dictionary on top of the stack: eg; type

```
LATEST ID.          ( will print a name )
: GODZILLA ;
LATEST ID.          ( displays GODZILLA )
```

8.3.2 Vocabularies

The vocabulary feature allows you to partition dictionary entries into named vocabularies. There are many good reasons to do this; for example, you can use the same names more than once in different vocabularies. If you have compiled a very large program using vocabularies, you can make subsequent compilation faster. Examples of commonly used vocabularies are SUPERFORTH and EDITOR : all the words described in this manual are contained in the SUPERFORTH vocabulary; the supplied Screen Editor is in an EDITOR vocabulary.

Words to handle vocabularies are (we will postpone examples until after these are described):

CONTEXT (--- ad) : a user variable which is used to determine which vocabulary is searched first of all, when words are interpreted or compiled.

CURRENT (--- ad) : a user variable which is used to specify the vocabulary in which new word definitions are appended. The definition of LATEST is, in fact:

```
: LATEST CURRENT @ @ ;
```

DEFINITIONS (---) : the compilation vocabulary is changed to be the same as the vocabulary which is searched first.

FORGET (---) is used in the form

```
FORGET <name>
```

to delete the dictionary entry for <name>, and all subsequent words, from the dictionary. A smart form of FORGET is provided which will detect if you FORGET through vocabularies and execution vectors: in the first case, SUPERFORTH is made the search and compilation vocabulary and a warning displayed; in the second case, the appropriate execution vectors are set

8.2 EXECUTION VECTORS

Execution vectors are used indirectly to execute other words: as such, they may be reassigned by the user to vary their effect. One use is for forward calls; ie; where you want to execute a word which has not yet been defined, an execution vector can be defined and then assigned to the word once it has been defined. The words to handle execution vectors are:

EXVEC: a defining word used in the form

```
EXVEC: <name>
```

to create an execution vector dictionary entry for <name>. By using ASSIGN and TO-DO the parameter field must subsequently be loaded with the compilation address of another compiled word, such that, when <name> is executed, this other word is executed. If an execution vector is used without having been assigned, an error message is output.

ASSIGN is used to define the word to be executed by an execution vector; it must be followed by a valid name in the input stream.

TO-DO is used with ASSIGN to define the word to be executed by an execution vector; it must be followed by the name of the word to be executed.

Example: type in the following sequence:

```
EXVEC: ANY-MESSAGE?
: RUDE-MESSAGE CR ." Push off " ;
: POLITE-MESSAGE CR ." Hello there " ;
ASSIGN ANY-MESSAGE? TO-DO RUDE-MESSAGE
```

now execute ANY-MESSAGE? by typing

```
ANY-MESSAGE?
```

which gives the response

```
Push off  ok
```

and reassigning ANY-MESSAGE? by

```
ASSIGN ANY-MESSAGE? TO-DO POLITE-MESSAGE
```

which changes the response to

```
ANY-MESSAGE?
```

to

```
Hello there  ok
```

Note that there are four words in the existing dictionary that are execution vectors, enabling a user to redefine their actions:

ABORT to enable a different abort sequence to be followed during a user-detected failure.

CLS to avoid trouble when outputting to a printer.

ERROR to help locate an error during compilation.

PROMPT which has already been used to execute .S

If an execution vector contains a forward reference, FORGETing through the forward reference will re-assign the execution vector to an error call. If this happens to PROMPT, simply type:

```
ASSIGN PROMPT TO-DO ok
```

To see this message, type MESSAGE COUNT TYPE

CREATE is used by the other defining words to create dictionary entries. For example, the definition of variable is:

```
: VARIABLE CREATE 0 , ;
```

An alternative version, which does not initialise the variable to zero, is:

```
: VARIABLE CREATE 2 ALLOT ;
```

DOES> (--- ad) is a word typically used in conjunction with CREATE to define the execution time action of a new user-specified defining word. It is used in the form
: <name1> ... CREATE ... DOES> ... ; to define a new defining word <name1> . When <name1> is used in the form

<name1> <name2>

it creates a new dictionary entry called <name2> which, when executed, leaves the parameter field address of <name2> as TOS and then executes the words following DOES> in the definition following <name1> . An example is a definition of the word CONSTANT:

```
: CONSTANT CREATE , DOES> @ ;
```

Now we can see what 99 CONSTANT FRED does:
when CONSTANT is executed, the TOS is 99. First of all CREATE is executed, which creates a new dictionary entry called FRED (because FRED is the next word in the input stream following CONSTANT). Then , is executed, which compiles the TOS (ie; 99) into the dictionary.
When FRED is executed, the address of the compiled 99 is left as TOS, and control now passes to the words following DOES> in the definition of CONSTANT. These execute @, which places the 99 as TOS and then ;, which terminates the actions of FRED . As you can see, this is exactly the action of a constant.

EXIT (---) is used in Compilation mode only, to prematurely terminate execution of a word. It does the same thing as the run time action of ; . EXIT must <u>not</u> be used within a DO ... LOOP or +LOOP or between a >R and R> pair, otherwise the system will almost certainly crash.

```
eg; : TEST BEGIN KEY DUP 32 =
                IF DROP EXIT THEN
                EMIT 0
           UNTIL ;
```

This enters an infinite loop: every time you press a key which is not a space, it is displayed on the screen. If it is a space, control returns to the keyboard.

the user variable FENCE .

14 unassigned execution vector
when an attempt is made to execute an execution vector which is not assigned to execute any other word. This may happen because it has not been initialised, or because the user has used FORGET to delete the word referred to by the execution vector from the dictionary.

16 division by 0
when an attempt is made to divide by zero.

17 division overflow
when integer division causes arithmetic overflow.

18 ROLL number negative
when TOS is negative on execution of ROLL.

19 ROLL beyond stack
when TOS is greater than the stack depth on execution of ROLL.

20 PICK number negative
when TOS is negative on execution of PICK.

21 PICK beyond stack
when TOS is greater than the stack depth on execution of PICK.

7.2 QL ERROR MESSAGES

In addition to the above error messages, many calls are made to the QL's ROM in the form of QDOS calls. On return to SUPERFORTH an error parameter is checked: if this is negative, a call to the QDOS error output routine is reported. The messages are as listed in the QL User Guide, in the Concepts Error handling section. Their error numbers are the negation of the numbers shown there.

7.3 USER DETECTED ERRORS

There are some SUPERFORTH words provided which carry out error checking and possibly invoke the sequence above:

?COMP (---) issues error message 1 if the system is not compiling.

?ERROR (flag n ---): if flag is TRUE, issues error message n and calls ERROR .

?EXEC (---) issues error message 2 if system is not executing.

?FOUND (n --- n) issues error message 8 if n is zero.

?STACK (---) issues error message 6 or 7 if stack is empty or full respectively.

CTRL <right>
deletes the character under the cursor.

F1 moves the line containing the cursor to the top line of the line store window. The line is deleted from the block but may be restored by the next command.

CTRL F1 copies the top line of the line store window to the line containing the cursor. The old cursor line and lines below it are moved down, and the last line lost.

F2 as F1, except that the second line of the line store window is used.

CTRL F2 as CTRL F1, except that the second line of the line store window is used.

CTRL SHIFT F1
deletes the line containing the cursor.

F3 requests another block (see notes 1 and 2).

CTRL F3 requests the next block in sequence: eg; if you are editing block 234, this requests block 235 (see note 1).

SHIFT F3 requests the previous block in sequence: eg; if editing block 234, this requests block 233 (see note 1).

F4 saves the block being edited to the default microdrive.

CTRL F4 renumbers the block being edited and saves it on microdrive (see note 2).

F5 creates a new, empty block (see notes 1 and 2).

SHIFT F5 marks the current block as not having been modified, so that it will not automatically be saved to microdrive, unless it is subsequently modified.

ENTER moves the cursor to the start of the next line.

CTRL <down>
clears the line store in the bottom window.

ALT T toggles a flag which indicates that character insertion and deletion acts over the current and next line. Cancel by ALT T again.

ALT M switches the default microdrive: you have to type in the new default number.

ESC exits from the Editor and returns to normal SUPERFORTH command mode.

CTRL SHIFT ESC
abandons the Editor and marks the block as not having been modified.

option that prints the current document to a file rather than a printer. You must have installed a printer driver, using INSTALL_BAS as described in the QL User Guide, which does not do anything except to output carriage return or line feed at the end of every line; that is, no preamble, postamble etc.
QUILL must be set up to print no header or footer on each page; tabs are acceptable. When your program is ready, print it to a file.

BLOCK (un --- ad): if not already present in the block buffer, BLOCK reads block un from mass storage.It then leaves on the stack the address, ad, of the first byte of the buffer in which the block is stored. If the buffer already held a block that had been updated, that other block is first saved on the default mass storage device.

BUFFER (un --- ad): assigns a block buffer to block un. If the buffer already holds an updated block, that block is saved. The address of the buffer, ad, is left on the stack. BUFFER may be used to create a new block, eg;
123 BUFFER DROP creates a new block numbered 123.

C/L (--- n): a constant representing the number of characters per line in a standard block. By convention, this is 64.

EMPTY-BUFFERS (---) unassigns the block buffer. An updated block is not written to mass storage.

FLUSH (---) performs the function of SAVE-BUFFERS and then unassigns the block buffer.

FLP (n ---) makes the default mass storage device floppy disk n.

FLP1_ (---) makes the default mass storage device FLP1_

FLP2_ (---) makes the default mass storage device FLP2_

L/B (---): a constant giving the number of lines in a standard SUPERFORTH block. By convention, this is 16.

LIST (n ---) lists block n on the display, using constants C/L and L/B to format the text.

LOAD (n ---) interprets and/or compiles SUPERFORTH source code from block n. It does this by saving the contents of BLK and >IN, which define the input stream. It then defines the new input stream by setting >IN to 0 and BLK to n. Block n is then interpreted or compiled until exhausted, when >IN and BLK are restored to their original values, thus returning to the original input stream.

MDV (n ---) makes MDV n the default mass storage device.

MDV1_ (---) makes MDV1_ the default mass storage device.

MDV2_ (---) makes MDV2_ the default mass storage device.

SAVE-BUFFERS (---) saves the block buffer to the default mass storage device, if the contents have been updated. The buffer remains assigned to the block.

THRU (un1 un2 ---) loads (as for LOAD) consecutively the blocks un1 to un2 inclusive.

3.12 FURTHER MEMORY HANDLING

Having considered the simple @ ! etc earlier on, we will now explain some more complex memory handling words.

First of all we list some words that must be used with extreme caution, since they could easily crash the QL if a mistake is made. These words are not standard FORTH-83 words but are QL specific. Most SUPERFORTH words use a 16 bit address to access memory within the SUPERFORTH dictionary (the address being relative to the start address of the SUPERFORTH dictionary). Sometimes it is necessary to access locations using an absolute 32 bit address, for example to write direct to a peripheral device or directly to the RAM used by the QL for the display.

A! (n dad ---): n is stored in absolute double-address dad , ie; like @, except that an absolute address is used

A@ (dad --- n): like @, except that an absolute address is used.

AC! (n dad ---): similar to C!

AC@ (dad --- n): similar to C@

A2! (d dad ---): similar to 2!

A2@ (dad --- d): similar to 2@

eg; to write to the display RAM:

```
HEX
: WHAT_A_MESS CLS 24000.       ( start at address HEX 24000 )
         1000 0 DO             ( write to HEX 1000 locations )
                2DUP           ( duplicate absolute address )
                I              ( store loop index )
                ROT ROT        ( get in form  n dad )
                AC!            ( write to display RAM )
                1 0 D+         ( increment absolute address )
              LOOP             ( and repeat )
         2DROP  ;              ( tidy up stack )
DECIMAL
WHAT_A_MESS
```

Now some standard SUPERFORTH words which are concerned with blocks of memory:

BLANK (ad un ---): un bytes of memory starting at ad are set equal to the ASCII character for space (decimal 32).

CMOVE (ad1 ad2 un ---) moves un bytes of memory from address ad1 to address ad2, moving the byte at ad1 to ad2 first, then proceeding towards higher memory.

CMOVE> (ad1 ad2 un ---): like CMOVE, except that the byte at address (ad1+un-1) is moved to (ad2+un-1) first, then proceeding towards lower memory.

3.11 NUMERIC CONVERSION

Words are provided to convert both from ASCII strings to integers and vice versa. These are used by the words described previously for input/output but are also available for the user.

The following words are used to convert from integers to ASCII characters and to format numbers prior to output to the display. An example of usage follows the definitions.

<# (---): initialise numeric output conversion. It sets up PAD for integer to string conversion.

# (d1 --- d2): d1 is divided by BASE and the remainder converted to an ASCII character which is then stored at the next lower address in PAD (see below) on the end of the string being converted. Both d1 and d2 must be positive double integers.

#S (d --- 0 0) converts d to a string of ASCII characters stored in PAD . If d is 0, then a single character 0 is appended to the string.

#> (d --- ad n) drops d and leaves the address and count of the string, formed by using # and/or #S in PAD . ad and n together are suitable for TYPE .

HOLD (n ---) saves the least significant byte of n in PAD as part of the output string being converted. Typically used between <# and #> .

SIGN (n ---): if n is negative, an ASCII minus sign is added to the start of the string in PAD, typically used between <# and #> after a number has been fully converted.

PAD (--- ad) leaves as TOS the lower address of a scratchpad area used to hold data for intermediate processing (typically before being printed on the display). It is used by all the standard words that convert and print numbers. The size of PAD is 84 bytes.

An example of the use of the previous words is to output an integer representing cents in dollars and cents format, eg; 1325 cents would be printed out as $13.25 (dollars to avoid problems with printers !). A word to do this is (assuming that the number of cents is TOS as a positive double-number and that BASE holds decimal 10)

```
: .DOLLARS <#         ( inialialise conversion )
            # #       ( convert and save 2 characters for cents )
            46 HOLD   ( save a decimal point character )
            #S        ( convert and save dollars characters )
            36 HOLD   ( save a $ character )
           #> TYPE    ( end conversion and print the string )
        ;
```

Try it with 13.25 .DOLLARS

SET_MODE (n ---): like SuperBASIC OVER, it sets the character printing mode in the current output window
n=0 is the normal mode
n=1 prints onto a transparent strip
n=-1 exclusive ORs the data onto the screen

TAB (n ---) moves the cursor to position n in the current line in the current window.

UNDER_ON (---) switches underlining on in the current output window. This only works in 8 colour mode.

UNDER_OFF (---) switches underlining off in the current output window.

3.10.4 Graphics handling

Some of these words need a list of floating point numbers to specify parameters: this is because the words are executed using calls to the QL's ROM. See Section 9 for information on the format of floating point numbers. Where the graphics origin is referred to, this means that the origin is at the bottom left corner of the current window and that the coordinates are scaled (just as they are for SuperBASIC: refer to the QL User Guide).

ARC (ad ---) draws an arc. ad is the address of a list of 5 parameters in this order (6 bytes each), which uses the graphics origin
angle subtended by the arc (radians)
Y coordinate of the end of the arc
X coordinate of the end of the arc
Y coordinate of the start of the arc
X coordinate of the start of the arc

BLOCK_FILL (n1 n2 n3 n4 n5 ---): like SuperBASIC BLOCK, this draws a rectangular block in the current output window. Pixel coordinates are used (origin at top left).
n1 is the colour
n2 is the width
n3 is the height
n4 is the X coordinate (top left corner)
n5 is the Y coordinate (top left corner)
The block is affected by the current printing mode (see SET_MODE). This is a much faster way of drawing horizontal and vertical lines, if the height or width is set to 1 respectively, than using LINE below.

BORDER (n1 n2 ---): like SuperBASIC BORDER, it sets the colour n1 and width n2 of the border of the current output window. If n1 is 128 the border is transparent.

CIRCLE (ad ---) draws a circle or ellipse, relative to the graphics origin. ad is the address of a list of 5 floating point parameters in this order:
rotation angle (radians)
radius of circle or ellipse
eccentricity of ellipse (1 for a circle)
Y coordinate of centre

((---) starts a comment in the input stream. It uses WORD to search for a) to terminate the comment. If the input stream is exhausted before a) is read, the search is terminated. Comment can be used freely, both in Interpretive and Compilation modes and to provide the means to document a program.

Now we can easily give some examples using these and COUNT TYPE as promised above. Type in this definition, not the comments in brackets:

```
: TEST CR             ( start on a new line )
   ." Type in up to 85 characters with several spaces"
                      ( message asking for input )
   CR QUERY           ( inputs up to 85 characters )
   SPAN @ .           ( print the number of characters )
   #TIB @ .           ( print number of bytes in TIB )
   >IN  @ .           ( print value of >IN )
   BEGIN              ( start a loop )
      32 WORD         ( read a word, space is delimiter)
      DUP C@          ( leave character count on stack )
      0 <>            ( flag = 0 if input exhausted )
   WHILE              ( only print a non zero string )
      CR COUNT TYPE   ( print the word just read )
      SPACE >IN @ .   ( print the value of >IN )
   REPEAT
   DROP CR ." No more input available" ;
```

Now type TEST and, after the message, type in several words separated by spaces (it doesn't matter what the words are at all since they are only printed, the dictionary is not searched for them. After you press ENTER to end the input, you should see each word printed on a new line followed by the latest value of >IN , which you should be able to match up with the input by counting characters.

To show the use of EXPECT, use the sequence
TIB 85 EXPECT instead of QUERY in TEST.
Try also using a number other than 85 , but less than 85 . You need a bigger input buffer to handle more than 85 characters.

3.10.3 Other screen commands

Several other words are provided to allow you to obtain various effects on the display. Wherever possible the same words as SuperBASIC keywords are used (in such cases, see the QL User Guide for explanation of some of the parameters). There are both text and graphics words. We will consider the graphics words in the next section.

AT (n1 n2 ---) moves the text cursor to column n1 and row n2 in the current output window. An error message is displayed if outside the window.

length of 255 characters. For example, the string HELLO would be stored in 6 bytes of memory, with the first byte holding the character count of 5, the second holding the value 72 (the ASCII code for H) and so on. The word ." described above stores the message in the dictionary in precisely this way. COUNT assumes the address of such a string is the TOS.

COUNT (ad --- ad+1 n) leaves the character count n, stored at memory location ad, as TOS and increments ad to leave ad+1 as 2OS. As can be seen, the stack is now in the correct state for TYPE . The sequence COUNT TYPE is commonly used. We will postpone an example of this until we have described a word called WORD below.

-TRAILING (ad n1 --- ad n2): ad and n1 are the address and character count of a character string. -TRAILING reduces the count by the number of space characters at the end of the string to leave a new character count n2. The string stored in memory is unchanged.

3.10.2 Keyboard input

Both single characters and words can be read from the keyboard.

KEY (--- n) leaves the ASCII code of the key pressed on the keyboard. KEY does not display a cursor, waits until a KEY has been pressed and does not display the character associated with the key. Words are provided to switch the cursor on and off (see below), eg;

```
KEY .      prints the ASCII code for the key
KEY EMIT   prints the character for the key
CURSOR_ON KEY  CURSOR_OFF EMIT  displays the
```

cursor and prints the character. If you do not want to wait and if no key has been pressed, the timeout can be adjusted (see section 10 on redirecting input and output).

KEYROW (n1 --- n2) leaves the value n2 of row n1 of the keyboard (see the QL User Guide).

Before considering word input , we will describe some user variables and the input buffer, which allow the user to manipulate input (examples follow below).

#TIB (--- ad): a variable containing the number of bytes read into the terminal input buffer TIB .

>IN (--- ad): a variable containing the present character offset within the input stream, ie; input from TIB or from microdrive or floppy disc. It shows how far the input scanner of the interpreter has reached.

SPAN (--- ad): a variable containing the actual number of characters read by EXPECT (see below)

TIB (--- ad) leaves the address of the terminal input

3.10 TERMINAL INPUT AND OUTPUT

This section describes words that read words or characters from the keyboard and print numbers or text onto the screen. As will be seen in a later section, these same words can be used to input or output text to other devices, eg; microdrives, printers etc.

3.10.1 Screen output

First of all, the words which output numbers to the screen: these all work on a number on the stack which is converted to characters according to the current base held in a variable called BASE . BASE is initially 10, which means that decimal numbers are input and output until you change its value. The first word is:

. (n ---) This prints out the TOS converted according to the value of BASE, followed by a single space: eg; (assuming BASE holds decimal 10)

```
123 .      prints out 123
-123 .     prints out -123
```

To see the effect of BASE, type in this sequence:

```
10 DUP . HEX .      which prints out    10 A
```

This is because the word HEX loads the value decimal 16 into BASE, which causes . to output TOS as a hexadecimal number.
This also causes input numbers to be treated as hexadecimal numbers, so now type in A BASE ! or DECIMAL, which both load decimal 10 into BASE.

Other words which print out numbers (all converted according to BASE) are as follows:

.R (n1 n2 ---) prints out n1 right aligned in a field n2 characters wide. If more characters than n2 are needed, then the whole number is printed as if . had been executed.
To see the effect, type (CR is explained below)

```
CR 123 5 .R
CR 123 6 .R
CR 123 2 .R
```

.S (---) prints out, non destructively, the contents of the stack as 16 bit integers. The TOS is printed to the right. .S has been explained before.

D. (d ---) prints out the double integer on top of the stack, followed by a single space, eg;

```
123.456 D.        prints 123456
-1234.56 D.       prints -123456
```

D.R (d n ---): like .R, except that a double-number is printed.

H. (n ---): like ., except that TOS is printed as a hexadecimal number. BASE is unchanged, eg;

```
49 H.     prints 31
```

3.9 CONTROL STRUCTURES

As in other languages you need to control the flow of your program: equivalent structures to SuperBASIC's IF ... THEN ... ELSE ... are provided in SUPERFORTH. These control structures can only be used in colon definitions: an attempt to execute them directly will result in an error message. These structures are:

IF ... ELSE ... THEN (flag ---): if the flag is true, the words between IF and ELSE are executed. Otherwise, the words between ELSE and THEN are executed.

```
eg;       : TEST IF ." True " ELSE ." False " THEN ;
now       0 TEST  prints out False
and       1 TEST prints out  True
```

BEGIN ... UNTIL (flag ---) UNTIL tests the flag and, if false, will then loop control back to BEGIN, to once again execute words between BEGIN and UNTIL. If the flag is true, then control passes to the words following UNTIL

```
eg;    : TEST 1 BEGIN DUP . 1+ DUP 10 >= UNTIL DROP ;
```

will, when executed, print out the numbers 1 to 10

BEGIN ... WHILE ... REPEAT (flag ---): this is another conditional looping structure. Here WHILE tests the flag which, if true, will execute the words between WHILE and REPEAT. If FALSE, it will branch to just beyond the REPEAT. When REPEAT is executed it branches back to BEGIN, eg;

```
: TEST 1 BEGIN DUP 10 <= WHILE DUP . 1+ REPEAT DROP ;
```

will again print out the numbers 1 to 10.

DO ... LOOP (n1 n2 ---): this is similar to a BASIC FOR loop. n1 is the limit of the loop index and n2 the starting value of the index. When LOOP is executed, the index is incremented and, if it has crossed the boundary between n1-1 and n1, the loop is terminated, eg;

```
: TEST 1 10 1 DO DUP . 1+ LOOP ;    will print out the
                                    numbers 1 to 9.
```

If n1 is the same as n2, the loop will be executed 65536, times because a DO ... LOOP is always executed at least once.

DO ... +LOOP : this is the same as DO ... LOOP, except that +LOOP uses the TOS to increment (or decrement) the loop index, eg;

```
: TEST 1 10 1 DO DUP . 1+ 3 +LOOP ;
```

will print out the sequence 1 2 3

```
: TEST 1 -12 -1 DO DUP . 1+ -5 +LOOP ;
```

will print out the sequence 1 2

@ (ad --- n) reads the location addressed by ad and leaves its value n on the stack
eg; FRED @ leaves 234 on the stack (assuming you have typed in the previous example).

2! (d ad ---): the double integer equivalent of !
eg; 987.654 2FRED 2!
writes the value 987654 into double variable 2FRED

2@ (ad --- d): the double integer equivalent of @
eg; 2FRED 2@ D. prints out 987654

C! (n ad ---) writes the least significant byte from the top of the stack into address ad:
eg; 99 FRED C! writes 99 into FRED

C@ (ad --- b) reads the byte addressed by ad
eg; FRED C@ leaves 99 on the stack.

Two other useful words associated with variables are:

? (ad ---) prints out the contents of location ad
eg; see below

+! (n ad ---) adds n to the contents of location ad and writes it back into ad

```
eg;        100 FRED !
           56  FRED +!
           FRED ?              prints out 156
```

3.7.2 Pre-defined constants

Some very commonly used constants are already compiled into the dictionary; these are:

0 1 2 3 -1 -2 and BL, which holds the value 32 (ie; the ASCII code for space or blank).

3.7 VARIABLES AND CONSTANTS

It is not always convenient or possible to use the stack, therefore variables and constants are provided. These are essentially the same as SuperBASIC variables, except that they must be created, using the SUPERFORTH words VARIABLE and CONSTANT, before they can be used.

```
eg;        VARIABLE FRED          creates a variable called FRED
           123 CONSTANT MARY      creates a constant called MARY
                                  which is assigned the value 123
```

CONSTANT assigns the number on top of the stack to the name following it. In strict FORTH-83, VARIABLE does not assign a value to the name following it, but SUPERFORTH assigns the value zero in such instances. When these new names are themselves executed, by typing them in, for example, a constant will leave its value on the stack and a variable will leave its address on the stack (note that this address is a 16 bit address in the SUPERFORTH dictionary, not an absolute QL address).

```
eg;        MARY .    will print out      123
           FRED U.   will print out an address depending on
                     FRED's location in the dictionary.
```

In a SUPERFORTH program you could, of course, use 123 instead of MARY, but you will often find it more meaningful to give a constant a name. If a particular constant is frequently used, giving it a name will save space in the dictionary.

It is possible to change the value of constants using a combination of ' or ['] and >BODY (see section 8).

There are also two more words for creating double integer constants and variables, 2CONSTANT and 2VARIABLE:

```
eg;        123.456 2CONSTANT 2MARY
           2VARIABLE 2FRED

now        2MARY D.               prints out 123456
and        2FRED U.               prints out a 16 bit address, just
                                  like FRED, but a different address
```

Discover for yourself whether we could have used JIM instead of 2MARY in the example above.

3.7.1 Using variables

The location of a variable is, in general, not much use on its own: other words are provided which write values to and read values from the relevant location. These are ! and @ respectively, their double integer equivalents 2! and 2@, and byte equivalents C! and C@ . These are defined as follows:

```
!          ( n ad --- ) loads the value n into the SUPERFORTH
                     dictionary location whose address is ad
           eg;       234 FRED !
           loads the value 234 into variable FRED ( don't forget
           that executing FRED left its address on the stack).
```

OR (un1 un2 --- un3): the bitwise logical OR of un1 and un2 is left as un3.
eg; 10 19 OR gives un3=27

NOT (un1 --- un2): un1 is inverted to give un2 (the one's complement is taken)
eg; 0 NOT gives un2=-1
-1 NOT gives un2=0

XOR (un1 un2 --- un3): the bitwise logical XOR of un1 and un2 is left as un3. This is useful for inverting selected bits:
eg; 15 6 XOR gives un3=9

3.5 STACK MANIPULATION

There are many words provided to manipulate numbers:

DUP (n --- n n) duplicates the TOS
eg; 123 DUP leaves two copies of 123
123 DUP + leaves 246 as the TOS

DROP (n ---) drops or loses the TOS
eg; 123 DROP leaves the stack unchanged

OVER (n1 n2 --- n1 n2 n1) duplicates the 2OS
eg; 1 2 OVER gives TOS=1, 2OS=2 and 3OS=1

SWAP (n1 n2 --- n2 n1) swaps the TOS and 2OS
eg; 1 2 SWAP gives TOS = 1 and 2OS = 2

ROT (n1 n2 n3 --- n2 n3 n1) rotates the 3OS to the TOS and moves the old TOS and 2OS down
eg; 1 2 3 ROT gives TOS=1, 2OS=3 and 3OS=2

PICK (... n1 --- ... n2) duplicates the n1th stack value, leaving the rest of the stack unchanged
eg; 5 7 3 2 1 0 4 PICK gives (--- 5 7 3 2 1 0 7)
5 4 3 2 1 0 1 PICK gives (--- 5 4 3 2 1 0 1)
0 PICK is identical to DUP
1 PICK is identical to OVER

ROLL (... n1 --- ...) rolls the n1th value on the stack to the top, moving all the intervening values down one place

eg; 5 4 3 2 1 0 4 ROLL gives (--- 5 3 2 1 0 4)
5 4 3 2 1 0 1 ROLL gives (--- 5 4 3 2 0 1)
2 ROLL is identical to ROT
1 ROLL is identical to SWAP

?DUP (n --- n n): duplicates the TOS if n is not zero
eg; 5 ?DUP gives (--- 5 5); 0 ?DUP gives (--- 0)

DEPTH (... --- ... n) leaves the number of 16 bit values on the stack as the TOS
eg; 1 2 3 4 DEPTH gives (--- 1 2 3 4 4)

MOD (n1 n2 --- n3) divides n1 by n2 to leave the remainder n3; the quotient is lost: eg;
136 30 MOD gives n3=16

NEGATE (n --- -n) negates n: eg;
543 NEGATE leaves -543 on the stack

3.2 DOUBLE LENGTH INTEGER WORDS

D+ (d1 d2 --- d3) adds double-numbers d1 and d2 to give the double_number result d3: eg;
123123. 234234. D+ gives d3=357357
Note that the display shown by the reassigned PROMPT gives d3 as two single integers 29677 5 . To see d3 type 123123. 234234. D+ D.

D- (d1 d2 --- d3) subtracts double-number d2 from d1 to give the double-number difference d3: eg;
123123. 234234. D- D. prints out -111111

DNEGATE (d --- -d) negates the double-number d: eg;
-123123. DNEGATE D. prints out 123123

3.3 OTHER ARITHMETIC OPERATIONS

The remainder of the integer arithmetic operations are described in this section.

*/MOD (n1 n2 n3 --- n4 n5): n1 is multiplied by n2 to give an intermediate 32 bit result, which is then divided by n3 to give the quotient n5 and remainder n4. That is, it is a combination of * and /MOD. The advantage of */MOD is that it retains an accurate intermediate result.
eg; 12 6 5 */MOD gives n5=14 and n4=2
and 10000 10 20 */MOD gives n5=5000 and n4=0.
In this latter example, typing
10000 10 * 20 /MOD gives an incorrect answer, since the intermediate result, 100000, is too big for a single length integer.

*/ (n1 n2 n3 --- n4): as for */MOD, except that only the quotient n4 is left on the stack.

UM* (un1 un2 --- ud): unsigned multiplication: ie; un1 and un2 are unsigned single length integers in the range 0 to 65535, and ud is an unsigned double length integer. un1 is multiplied by un2 to give a double length product ud:
eg; 35000 100 UM* D. prints out 3500000

UM/MOD (ud un1 --- u2 u3): unsigned division: ie; unsigned double-number ud is divided by un1 to give quotient u3 and remainder u2, both unsigned:
eg; 123456. 9999 UM/MOD gives u2 = 3468

integers must not be preceded by a + sign.

2.4 NAMES OF SUPERFORTH WORDS

Names of SUPERFORTH words may contain any ASCII character (excluding control characters or the space character) or the additional characters of the QL (eg; greek characters). Upper case letters are distinguished from lower case: eg; FRED, fred and Fred are treated as three different names. The space and control characters are used to separate words and the user must be particularly careful about the use of spaces: eg; -123 and - 123 mean two entirely different things (the first is an integer -123 and the second is the subtract operation followed by the integer 123).

2.5 THE STACK

A fundamental concept in SUPERFORTH, and in computing in general, is the stack. All arithmetic operations use numbers on the stack. A stack can simply be viewed as a pile of numbers: eg; consider a series of numbers, similar to that described in the previous section, for which each number in turn could be written on a piece of paper and then stacked on a table. The SUPERFORTH stack is a similar structure maintained in the memory of the QL. Usually only the last two numbers entered, called the top of the stack (TOS) and the second on the stack (2OS), are available for arithmetic operations. A stack can also be described as a "last in first out" data structure.

If we enter two numbers, eg; by typing 123 234, then the TOS is 234 and the 2OS is 123. If we want to add these two numbers we type +, which, as will be seen later, adds the TOS to the 2OS and leaves the result as the new TOS (the original TOS and 2OS are lost) ie; the stack now contains only 357. To see this in action type

```
          123 234 + .  <ENTER>
```

(where <ENTER> means press ENTER)
which gives the response

```
                    357  ok
```

The word . tells SUPERFORTH to print out the value of TOS on the output display. The output ok is simply SUPERFORTH's way of saying that it has carried out the operation and is now waiting for more input.

To see this more graphically, type

```
          ASSIGN PROMPT TO-DO .S  <ENTER>
```

(this will be explained later) and then type

```
          123    <ENTER>
          234    <ENTER>
          +      <ENTER>
```

and after each ENTER you will see the contents of the stack printed on the display, the TOS to the right. It is a good idea always to do this when working through examples or debugging new SUPERFORTH definitions. The remainder of this user guide will

1.4 INPUT FROM THE KEYBOARD

Commands, numbers and new definitions may be entered at the keyboard simply by typing them in; SUPERFORTH words must be separated by at least one space character. The line is not processed by SUPERFORTH until the ENTER key has been pressed: before pressing ENTER, the line may be edited using the left and right arrows and CTRL, exactly as if entering a BASIC program. When the line has been entered, SUPERFORTH executes or compiles each SUPERFORTH word in turn and, when complete, outputs the message ok (unless there have been errors) and waits for more input: eg; try typing the following (note the spaces between 1,2,+ and .):

```
1 2 + .
```

This will cause the response

```
3  ok
```

The line input buffer will accept up to 85 characters; if more than 85 are entered, then they will be processed without ENTER having been pressed: the last word may be a part word, which may cause an error.

1.5 INPUT FROM MICRODRIVES AND FLOPPY DISCS

SUPERFORTH has a particular way of handling input from mass storage, which in QL terms means microdrives or floppy disks. This will be fully described in Section 4, but for now it is sufficient to say that the SUPERFORTH interpreter/compiler still sees this input as a stream of SUPERFORTH words.

1.6 BACKING UP THE SYSTEM

To make a backup copy of QL SUPERFORTH, place the supplied microcartridge in drive 1 (the left hand drive) and a fresh microcartridge (which need not be formatted) in drive 2. Then enter LRUN MDV1_BACKUP. To make a backup on floppy disk, use the utility for file transfer supplied with your floppy disk interface - SUPERFORTH is device name driven and will transfer and run without any problems at all (ignore any 'flp bad name' error message that might be displayed after the editor is loaded from a floppy - it is just reporting that an unusual name has been encountered, and will automatically adjust itself to cater for the new default device).

NOTES: (1) A Quill file called UPDATES_DOC may be present on the cartridge supplied. It is the policy of Digital Precision to continually improve & refine its software - the file will contain a list of updates to the system and should be read in conjunction with this manual.

(2) Sinclair, QL & SuperBASIC are trademarks of Sinclair Research Ltd.

(3) This manual is designed to fit into your User Manual file as supplied with your QL.

APPENDIX TO THE SUPERFORTH MANUAL

1>> Revised instructions for REVERSI

The following supersedes Chapter 12 of the manual.
To run the program, take a reset QL & EXEC the file REVERSI. If the device is mdv1_, then the appropriate command is:

```
EXEC MDV1_REVERSI
```

Once the load screen appears, press CTRL & C simultaneously.

The aim of Reversi (also called Othello) is to end up with the most pieces on the 8x8 board. You & your opponent make moves alternately, using pieces which are black on one side & white on the other. The player who is black will always place them with black facing up, etc.

To make a move, you must place a new piece such that you trap one or more of your opponent's pieces between the new piece & one or more of your own pieces, in one or more continuous (ie; no intervening vacant squares) straight lines along rows, columns or diagonals. You can only play on a vacant square - hence a game can never last more than 64 moves excluding passes (you "pass" if you cannot make any move - it is then your opponent's turn). The move is completed by changing all the trapped pieces to your own colour (ie; by flipping them). If this sounds complex don't worry - SUPER REVERSI will not permit illegal moves, so by playing you will soon pick up the game. Remember - a move must result in at least one flip.

The game is usually started with four pieces placed in the centre (as shown when you run the game), but SUPER REVERSI gives you the option of setting up your own starting position. Black always moves first - you are given the option at the beginning of the game to be either Black or White. Do not jump to the conclusion that the first player has an advantage - Reversi is far more subtle than that!

The game finishes when neither player can move. The player who then has the most pieces showing on the board is the winner (draws are possible) - SUPER REVERSI keeps track of the number of pieces for each side throughout the game. Note that it is only at the final position that the number of pieces decides the outcome - earlier on, it is not necessarily good strategy to maximise the number of pieces of your colour (to do so would give your opponent more pieces to flip over later). Naturally, you must have at least one piece on the board or else you will have to pass for the rest of the game.

You make a move either by typing the grid reference of the move: H3 or 3H followed by ENTER or SPACE, or by moving the cursor to the square using the cursor keys & pressing ENTER or SPACE.

Any other key either selects one of the command options listed below or cancels the move.

O .... Display Options
M .... Mode - QL vs QL, Human vs Human or Normal
X .... Exchange sides (ie; cheat!)
S .... Setup a new position
R .... Retract one or more moves (ie; cheat!)

?	Hint - Suggest a move (ie; cheat!)
Q	Quit ie; Resign
L	Change skill level
W	Redraw screen
T	Toggle sound on/off (when QL moves)
ESC then CTRL+C	Return to SuperBASIC

Note that the O option makes the above table redundant !

When the QL is thinking you can interrupt it by holding down the I key until it makes a move, this is useful if you have accidentally selected a high playing level and do not wish to wait for the QL to finish it´s deliberations (or again, of course, to cheat).

There are 9 playing levels ranging from the easiest at level 1 to level 9 (where the QL has a 7 move look-ahead !). On levels 3 & above, while the QL is thinking, it displays the move currently being examined, the best move so far & the last move considered: with these last two it also displays a value which indicates how good the move is (the higher the value the better for the QL). This makes waiting for the QL to move quite interesting. You can change levels during the middle of a game. Levels (& approximate times taken) are:

1	Beginner	0.1 seconds
2	Novice	2 seconds
3	Intermediate	30 seconds
4	Fairly strong	1 minute
5	Strong	2.5 minutes
6	Very Strong	5 minutes
7	Master	10 minutes
8	Expert	30 minutes
9	Champion	1.5 hours

When a move is made, the pieces affected flash in red for a few seconds.

If you want to see the computer play itself (different levels for each side possible !) use the M option & choose Q. If you want the QL to supervise a game between two humans, choose M followed by H .

SUPER REVERSI is a multitasking SUPERFORTH program. That means you can run it at the same time as other tasks (M68000 programs, SUPERCHARGED programs, EXEC-able files output by other compilers, the SuperBASIC task or other SUPERFORTH programs - including other ´copies´ of SUPER REVERSI itself'). Use the W option to redraw the screen if it appears untidy while multitasking. Naturally, if you run multiple copies of SUPER REVERSI they will all run slow. CTRL+C allows you to page freely between SUPER REVERSI & SuperBASIC.

The Setup option (S) is useful either to solve Reversi problems or to return to an abandoned position. Any position may be setup but some may not be much fun (ie; the empty board, which will of course result in a drawn game!). The keys to be used for setup are displayed on the setup screen:

Arrow Keys	Move cursor
W	Put a White piece on square
B	Put a Black piece on square
C	Clear board
N	Clear square
Esc	Terminate setup mode

Use the I option if you want the QL to move immediately - but remember you are handicapping it by denying it the agreed time for the move. You must hold the I key down for a few seconds until the red pieces are displayed - the keyboard is only polled intermittently to keep things fast.

Here are some tips that should improve your playing strength:

(a) Do not 'grab' material - position is more important than material until the last stages of the game.

(b) In the beginning of the game, try to stay within the central 4x4 square area. The first player to move out of this area is often at a disadvantage.

(c) The most valuable squares are the corner squares as once occupied their occupier can (obviously) never be flipped. If the loss of a corner is inevitable then play should be directed towards blocking its effectiveness (eg; the corner A1 is much less useful for Black if Black also has A3 & White has A2).

(d) Edge squares other than corners are somewhat dangerous to occupy, especially those immediately adjacent to corner squares. They can provide an avenue of attack for your opponent culminating in his occupation of a corner square.

(e) At every stage of the game try to make moves that, while not contradicting (a)-(d) above, reduce the number of options open to your opponent to a minimum.

(f) Long diagonals are useful only if a corner on that diagonal has been secured, or if the diagonals are for some other reason immune from attack.

(g) Remember not to count on your opponent making oversights!

SCORE INTERPRETATION (assuming 64 pieces are on the board)

32-32	Drawn
33-31 to 35-29	Narrowly won
36-28 to 41-23	Comfortably won
42-22 to 49-15	A Smashing victory
50-14 or better	!?!!

2>> BLUDNERS!

(a) The SUPERFORTH word SUPERFORTH (8.3.2) should have been FORTH

(b) the BEGIN ... UNTIL example (3.9) will print out from 1 to 9, not 1 to 10

(c) the second DO... +LOOP example (3.9) prints out the sequence 1 2 3, not 1 2

(d) the definitions of TEAMS (8.3.2) should both have quotes immediately after the word men

(e) in 5.5, if you want 64 characters/line you must edit block 3 to set the character size in windows #1, #2 and #3 - these are set up in the definition SET_PARS

(f) The page beginning with 3.8 in the manual has had holes punched on the wrong side, putting 3.7.2 after 3.8 - to rectify, flip the page

3>> SUPERFORTH Version 2.0

Following our policy of ever improving our product (difficult though this is !) version 2.0 has many enhancements. The principal enhancement involves string handling, which is described in detail in part 6>> of this appendix. The other improvements are given below.

(a) Two SUPERFORTH definitions have been added to the dictionary to give you the option of using upper or lower case letters to execute SUPERFORTH words. They are:

LOWER   changes mode so that standard SUPERFORTH words will be executed when lower or upper case letters are typed in, eg;

```
                    LOWER dup DUP       will execute DUP twice
```

UPPER   reverses the effect of LOWER, eg;

```
                    UPPER dup           will give an error.
```

When in LOWER mode, lower case definitions are inserted into the dictionary in upper case form.

(b) To avoid infinite loops due to enhancement (a) the default words executed by the execution vectors CLS, ERROR and ABORT (8.2) are changed to (CLS),(ERROR) and (ABORT). Previously they were lower case equivalents.

(c) In the screen editor an extra command has been added: ALT+F or ALT+f select floppy disk as the default drive.

4>> New Utility Blocks

Another 4 utility blocks have been included. These are:

(a) Block 6, VLIST

Contains a definition of VLIST which lists all the words in the current vocabulary on the current output device, 8 words on a line. Type:

```
         6 LOAD  VLIST
```

(b) Block 7, TURNKEY

This enables you to create a stand-alone EXECable SUPERFORTH program, ie; it will run as a separate, dedicated task. (SUPER REVERSI was generated in this way). To use it first of all load your SUPERFORTH application from SUPERFORTH blocks or other file, then type:

```
          7 LOAD
and       TURNKEY <name>
```

where <name> is the SUPERFORTH word you wish the stand-alone task to execute (REVERSI in the case of SUPER REVERSI). Then follow the instructions on the screen.

eq: if you wanted a task to print out numbers 0 to 999 and then terminate, first define a word. (Note the suicide word BYE at the end which must be included to terminate the task)

```
          : SIMPLE_EXAMPLE  1000 0 DO I . LOOP BYE ;
then      7 LOAD
then      TURNKEY SIMPLE_EXAMPLE
```

To execute your new task from SuperBASIC or SUPERFORTH type

```
         EXEC MDV1_filename
```

In block 7 there is a word defined called DENAME which erases all the SUPERFORTH headers. If you develop a program for sale then we must insist that you use it to prevent you inadvertently selling a SUPERFORTH system as well.

(c) Block 8, LOAD_BIN

For machine code programmers we have included a way of loading machine code generated by conventional assemblers (eg; Metacomco's). To demonstrate how to use this we have included two other files:

(1) "mdv1_example_asm"

```
and       (2) "mdv1_example_bin"
```

where the second is an assembled version of the first. These give you three new code definitions:

```
NOR  ( n1 n2 --- n3 )  n3 is the logical NOR of n1 and n2
3*   ( n1 --- n2 )  n2 is n1 times 3
3DUP ( n1 n2 n3 --- n1 n2 n3 n1 n2 n3 ) equivalent to
                      2 PICK 2 PICK 2 PICK
```

To load these, type:

```
                       8 LOAD
and                    LOAD_BIN mdv1_example_bin
```

then try them out.

A complete description of the assembler format needed is given in "mdv1_example_asm". This format must be followed to ensure a correct binary file is assembled. Also study chapter 11 of the manual.

(d) Block 9, CREATE_DEVICE

For those with floppy disks which are not referred to as flp we have included a way for you to define your own default device, eg;

```
type        9 LOAD
            CREATE_DEVICE FDK1_
```

Now if you type FDK1_ it will become the default device for handling standard SUPERFORTH blocks. You can return to mdv by typing MDV1_ .

(e) An example of how to use SUPERFORTH graphics has been supplied in a file called CIRCLE_FTH - use the editor to examine it.

5>> Transferring SUPERFORTH to another device

To save SUPERFORTH V1.6 onto another device (eg; floppy disk), do NOT use a copying utility as suggested in 1.6: instead use LRUN MDV1_BACKUP and choose option 'E'. Just follow the prompts. Note that SUPERFORTH may be started independently of BOOT by typing:

```
            EXEC MDV1_FORTH83_JOB
or          EXEC FLP1_FORTH83_JOB   as appropriate.
```

6>> String handling

A set of powerful string handling words have been added to SUPERFORTH version 2.0 to give you the same sort of operations that are available in SuperBASIC, but, of course, much much faster.

Storage Of Strings

Strings are stored as a sequence of characters, one character per byte. The characters are preceded by two other bytes, the maximum permitted length of the string and the actual length of the string. Because the numbers are stored in bytes the maximum possible length of string that may be specified is 255 bytes. For example if a string called MONTH, with a maximum length of 9 characters, contains the value "January", it will be stored in this form:

Address	Value	Meaning
n	9	Maximum length
n+1	7	Actual length
n+2	74	Character "J"
n+3	97	Character "a"

n+4	110	Character "n"
n+5	117	Character "u"
n+6	97	Character "a"
n+7	114	Character "r"
n+8	121	Character "y"
n+9	?	Not used
n+10	?	Not used

When MONTH is executed (see below), it leaves the address of the actual length byte on the stack. The contents of MONTH may be printed, as for any SUPERFORTH string, by using COUNT TYPE (see section 3.10.1 in the SUPERFORTH manual). For example, given MONTH as above, typing:

```
        MONTH  COUNT  TYPE   will display   January
```

Defining Strings

STRING (n1 ---) is used in the form

```
                    n1 STRING <name>
```

to create a dictionary called <name> which, when executed, will leave the address of it's length byte on the stack. The value n1, which must be on the stack, defines the maximum length, in bytes, of the string. Initially the string is empty. Eg; type:

```
                    10 STRING MONTH
```

to create an empty string called MONTH, which may be loaded with a maximum of 10 characters.

STR_ARRAY (n1 n2 ---) is used in the form

```
                    n1 n2 STR_ARRAY <name>
```

to create an array of strings called <name>. This array contains n1 strings, each with a maximum size of n2 characters. <name> may later be executed in the form: n3 <name>

which will leave the address of the length byte of the (n3-1)th string in the array, on the stack. Using n3=0 will give the address of the first string in the array. If n3 >= n2 an error will occur with the message

"String index out of range"

Eg; type:

```
                    7  9  STR_ARRAY  DAYS_OF_WEEK
```

to create an array called DAYS_OF_WEEK with 7 strings, each with a maximum of 9 characters.

Load the first array element with:

```
                    0 DAYS_OF_WEEK READ" Sunday"
```

Read it with:

```
                    0 DAYS_OF_WEEK COUNT TYPE
```

Similarly

```
                    5 DAYS_OF_WEEK READ" Friday"
```

STR_CONST (---) A defining word used in the form:

```
                    STR_CONST <name> "<character string>"
```

Creates a dictionary entry called <name> which, when later executed, leaves the address of the string's actual length byte on the stack. The following character string must be surrounded by a pair of " characters. The maximum length byte is set equal to the actual length byte. Eg;

```
                    STR_CONST PRAISE "SUPERFORTH is great"
                    PRAISE COUNT TYPE
```

Data Input To Strings

INPUT (ad1 ---) Reads a line of text from the current input stream into the string at ad1, eg assuming string MONTH is defined as above, type (<enter> means press the ENTER key):

```
MONTH INPUT <enter> January <enter>
```

which will load the word January into MONTH. You can prove this by typing:

```
MONTH COUNT TYPE
```

READ" (ad1 ---) In interpretive mode.
(---) In a colon definition (compilation mode).

Reads the following characters, up to but not including the next " character or <enter>. In interpretive mode it assigns these characters to the string at ad1 eg;

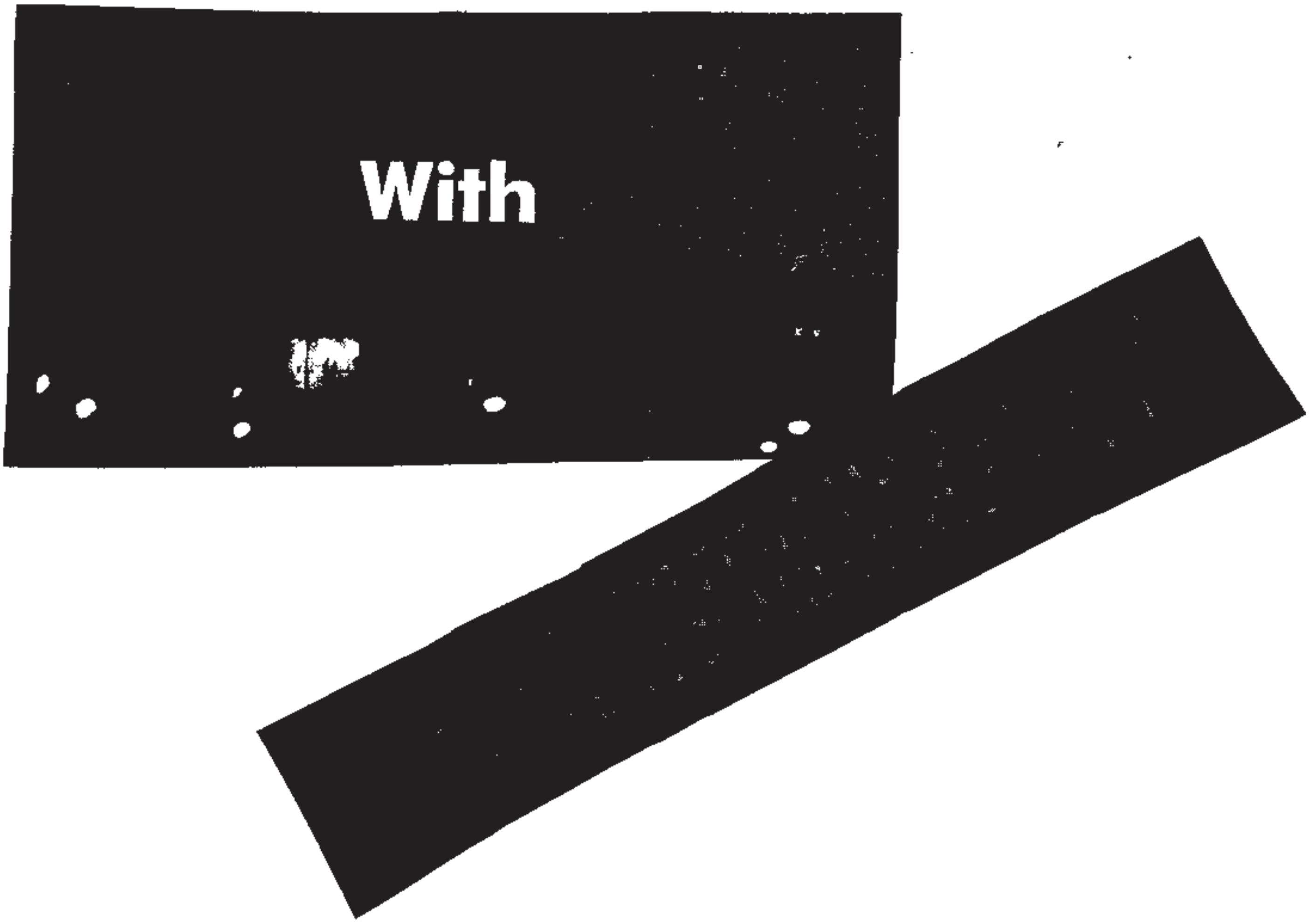
```
MONTH READ" February"
```

In compilation mode these characters are inserted into the colon definition as a constant string. When the colon definition is later executed, the address of the actual length byte of this constant string is left on the stack. This string may then be used for any operations described below. Eg;

```
: TEST READ" An example" COUNT TYPE ;
TEST
```

This behaves just like:

```
: TEST ." An example" ;
TEST
```



String Characteristics

LENGTH (ad1 --- n1) Leaves the actual length, n1, of the string at ad1 on the stack eg; (assuming MONTH holds February)

```
                    MONTH LENGTH .        prints  8
```

MAX_LEN (ad1 --- n1) Leaves the maximum length, n1 , of the string at ad1 on the stack eg;

```
                    MONTH MAX_LEN .       prints  10
```

UNUSED (ad1 --- n1) Leaves the number of spare bytes, n1, in the string at ad1 on the stack eg;

```
                    MONTH UNUSED .        prints  2
```

String Operations

Examples of the use of the following words are given in the next section.

APPEND (ad1 ad2 ---) Appends the string at ad1 onto the end of the string at ad2.

APP_CHAR (n1 ad1 ---) Appends the character whose ASCII value is n1 onto the end of the string at ad1.

CHAR (ad1 n1 --- n2) Leaves on the stack the ASCII value, n2, of the character at position n1 in the string at ad1.

CLEAR (ad1 ---) Sets the actual length of the string at ad1 to zero.

INSERT (ad1 ad2 n1 ---) Inserts the string at ad1 into the string at ad2 at position n1. The end of the string, from position n1 upwards, is moved up by the number of characters in string ad1, and the length of string ad2 adjusted accordingly.

INS_CHAR (n1 ad1 n2 ---) Inserts the character, whose ASCII value is n1, into the string at ad1, position n2. Characters from position n2 are moved along 1 position in string ad1.

INS/DEL (ad1 n1 n2 ---) If n2 is positive INS/DEL moves the string ad1, from position n1 upwards, along by n2 positions, and increases the length by n2. If n2 is negative, then -n2 characters are removed from position n1 upwards.

LOCATE (ad1 ad2 n1 n2 --- n3) The string at ad1 is a pattern, LOCATE searches the string at ad2, from position n1 upwards, for the first occurrence of the pattern. If found then n3 holds the start position of the matching characters, otherwise if the search failed, n3 is zero. If n2=0 the search is dependent on the case of alphabetic characters (ie "A"<>"a"), if n2<>0 the search is case independent (ie "A"="a").

LOC_CHAR (n4 ad2 n1 n2 --- n3) Like LOCATE except that the pattern is a character whose ASCII value is n4.

LOSE (ad1 n1 n2 ---) n2 characters are deleted from the string at ad1, position n1. Characters at the end of the string are moved down and the length decreased by n2.

REPLACE (ad1 ad2 n1 ---) The characters at position n1 upwards in the string at ad2 are replaced by the contents of the string ad1. The length of string ad2 is unchanged.

REPL_CHAR (n1 ad1 n2 ---) The character at position n2 in the string at ad1 is replaced by the character whose ASCII value is n1. The length of string ad1 is unchanged.

SLICE (ad1 n1 n2 ad2 ---) n2 characters are copied from position n1 in the string at ad1 into the string at ad2. The previous contents of string ad2 are lost. String ad1 is unchanged.

TAKE (ad1 n1 n2 ad2 ---) TAKE is like SLICE, except that the characters in the string at ad1 are removed from string ad1 and the length adjusted accordingly.

TAKE_CHAR (ad1 n1 --- n3) The character at position n1 in the string at ad1 is removed from string ad1 and has it's ASCII value left on the stack.

UP_CHAR (n1 --- n2) If the character, whose ASCII value is n1, is a lower case character, it is converted to an upper case character with value n2, otherwise n2=n1.

String Comparisons

$= (ad1 ad2 --- flag) The flag is true if the string at ad1 is equal to the string at ad2. The case is significant ie "A"<>"a".

$== (ad1 ad2 --- flag) Like $= except that the comparison is case independent ie "AbC" is equal to "aBC"

$< (ad1 ad2 --- flag) The flag is true if the string at ad1 is less than the string at ad2. The comparison is type 2 as described in the QL User Guide (Concepts - String comparison) ie case dependent with embedded number strings compared as numbers.

$> (ad1 ad2 --- flag) The flag is true if string ad1 is greater than string ad2. The comparison is as described in $<.

C== (n1 n2 --- flag) The flag is true if the character with ASCII value n1 is equal to character n2. The comparison is case independent ie "A"="a".

COMPARE (ad1 ad2 n1 --- n2) Compares the strings at ad1 and ad2. n1 defines the type of comparison, n1=0 is type 0, n1=1 type 1, n1=2 type 2 and n1=3 type (QL User Guide Concepts - String comparison). n2=0 if the strings are equal; n2=-1 if string ad1 < string ad2; n2=1 if string ad1 > string ad2.

Illustrative Examples

The use of the above words will be demonstrated by typing in the following (do not bother to type in the explanatory comments):

```
50 STRING NAME                  ( A string to hold the full name )
20 STRING CHRISTIAN                       ( The christian name )
20 STRING MIDDLE                ( The middle name )
20 STRING SURNAME               ( and the surname )

( First two words to save typing )
: $. COUNT TYPE ;  ( Use is eg. CHRISTIAN $. to print a string )
: ASCII BL WORD 1+ C@ ;         ( Gets the ASCII value of the next)
                                ( character in the input stream )

( Now start loading the strings, <enter> means press ENTER )
SURNAME INPUT <enter> Clark <enter>
( Clark gets loaded into SURNAME, try SURNAME $. )

( Copy it into the full name string )
SURNAME NAME APPEND                       ( try NAME $. and SURNAME $. )

( Oops, we meant to have an e at the end )
ASCII e NAME APP_CHAR           ( sticks an e on the end )

CHRISTIAN INPUT <enter> Ann <enter>
( Loads Ann into string CHRISTIAN )
MIDDLE INPUT <enter> Rosemary             <enter>
( and this goes into string MIDDLE )
( Prove these by CHRISTIAN $. and MIDDLE $. )
( Note READ" could have been used instead of INPUT )

( Insert the christian name into NAME )
CHRISTIAN NAME 1 INSERT

( Try NAME $. , we need a space inserted, so ... )
BL NAME 4 INS_CHAR              ( see manual 3.7.2 for BL )

( Do the same for the middle name )
BL NAME 4 INS_CHAR
MIDDLE NAME 5 INSERT            ( Try NAME $. )

( Now suppose we want the middle name to be Mary )
NAME 5 4 LOSE                             ( gets rid of Rose )
NAME 5 CHAR UP_CHAR NAME 5 REPL_CHAR      ( changes m to M )

( To demonstrate the difference between SLICE and TAKE )
NAME 5 4 MIDDLE SLICE           ( do NAME $. and MIDDLE $. )
MIDDLE CLEAR                    ( clear MIDDLE, try MIDDLE $. )
NAME 5 4 MIDDLE TAKE            ( do NAME $. and MIDDLE $. )
```

```
( Lose the superfluous space )
NAME 5 TAKE_CHAR DROP        ( or NAME 5 1 LOSE )

( To replace Ann with Sue )
CHRISTIAN READ" Sue "        ( Note the space after Sue
CHRISTIAN NAME 1 REPLACE     ( type NAME $. )

( And to insert a middle name )
MIDDLE READ" Lucy "
MIDDLE NAME 5 INSERT         ( type NAME $. )

( To locate the position of a name try )
MIDDLE NAME 1 1 LOCATE .     ( Prints the position of Lucy )

( Finally a colon definition which shows how to split up a )
( string such as NAME into it's individual parts. )

: GET_NAMES
   BL NAME 1 0 LOC_CHAR      ( Find position of first space )
   ?DUP
   IF                        ( Have located a space )
     DUP 1+ BL NAME
     ROT 1 LOC_CHAR ?DUP     ( and position of second space )
     IF                      ( Have located another space )
       SWAP NAME 1 2 PICK 1- ( set up to read christian name )
       CHRISTIAN SLICE       ( and copy it into CHRISTIAN )
       2DUP - 1- NAME ROT 1+ ROT ( Set up to read middle name )
       MIDDLE SLICE                      ( and copy it into MIDDLE )
       NAME SWAP 1+ NAME LENGTH  ( Set up for surname )
       1+ OVER - SURNAME SLICE   ( and copy into SURNAME )
       CR NAME $.               ( To see results do this ... )
       CR CHRISTIAN $.          ( and this etc. )
       CR MIDDLE $.
       CR SURNAME $.
     ELSE
       CR ." No middle name available"
     THEN
   ELSE
     CR ." No first name available"
   THEN    ;

( Now try eg. )
GET_NAMES                              ( With NAME as above )
NAME READ" Johann Sebastian Bach"
GETNAMES
NAME READ" Fred Smith" GET_NAMES
```

New Words To Use Strings

DELETE_FILE (ad1 ---) deletes the file whose name is contained in the string at ad1.

DEVICE_STATUS (ad1 --- n1) Returns the status of the device or file whose name is contained in the string at ad1. If the device is valid and a file does not exist then n1 is zero. If n1=-8 the file already exists. For other values of n1 see the QL manual, concepts - error handling. (the codes there are -n1, eg if n1=-7, look at error 7)

OPEN_DEVICE (n ad1 --- d) Opens a channel to the device whose name is contained in the string at ad1. n and d are the same as for OPEN (See SUPERFORTH manual 10.1) Eg;

```
20 STRING FILE
FILE READ" mdv1_example"
2 FILE OPEN_DEVICE ( create a file called mdv1_example)
CLOSE                 ( close the channel )
FILE DELETE_FILE      ( and delete the file )
```

STATUS (--- n1) Takes the next word in the input stream, assumes it is a device or file name, and tests it's status. n1 is the same as for DEVICE_STATUS. Eg;

```
STATUS mdv1_example
gives n1=0 if the file does not exist
      n1=-8 if it already exists
      etc
```

Error detection

In the string operations, if the specified string or substring is too big then an error will be detected and an appropriate message printed on the output device. Possible messages are:

```
String too long
String size too big
String index out of range
```

This action may be redefined using the techniques described in 8.2 in the SUPERFORTH manual.