IS – Forth Manual

BY INTELLIGENT SOFTWARE LTD

INTRODUCTION – FORTH – What is it and who wants it?

0.0 Forth – All Things To All People ?

Forth is a revolutionary, very fast, portable and adaptable language which is becoming very popular for games programming, control, general-purpose applications and is being tipped by many to replace BASIC. It, not only, combines the advantages of an assembler, compiler, interpreter, and operating system, but actually allows programmers to add to the language and so to adapt it to individual needs and purposes. Forth may even be used to produce other, derivative, languages.

0.1 History and Origins.

Forth was developed by an American, Charles Moore, during the 1970's. Moore was working with computers, on the control and data processing of a radio-telescope, in California, USA. He required a fast, compact, flexible and extendable computer language to increase the through-put of his machine. Finding that nothing suitable for his purposes was available, he set about writing a fourth generation (as he put it) computer language. The file handler on the system, that he was using, only allowed five character titles and so "Fourth" became abbreviated to the familiar "Forth". It was developed specifically to increase the productivity of the programmer, without sacrificing the efficiency of the computer.

Most other computer languages had suffered from serious limitations eg. assembler is fast – but can only be used for one type of processor, is difficult to learn, relies upon intimate knowledge of the system hardware and has to be relearned if one needs to move to another type of computer. Basic, however, is easy to learn and is available on most processors but is too slow for many applications and has many dialects. Forth, being much faster than Basic, easier to use than Assembler, and available for almost all processors had and has great appeal.

Keen enthusiasts were soon spreading the news, by word of mouth, throughout the radio-astronomy and scientific fraternities of America's west coast. Moore and some colleagues, seizing a good opportunity, left their posts to form Forth Inc. who produce commercial implementations and software packages. "Forth" is a trade-mark of Forth Inc.

Soon, there were 'colonies' of users across the U.S.A. and Europe who had bought and extended or written their own versions and a need for some conformity was becoming apparent. Two standardizing bodies emerged.

0.2 Fig-Forth.

The Forth Interest Group Inc. are an international group of users who banded together to produce their own Forth standard called Fig-Forth. They
use as their reference the Forth Installation Manual (a publication of the group). Applications written in Fig-Forth may be converted to run under other Forths fairly easily, however, there are some differences which will not be investigated here, to avoid later confusion. Users of IS-FORTH would be well advised to avoid books etc. describing Fig-Forth, in the early stages.

0.3 Forth-79 and Forth-83.

Following an attempt to promote a standard by the European Forth User's Group (EFUG), Forth-77 was born. As the language evolved, an interim version (Forth-78) was published by the Forth Standards Team (An international body of users and manufacturers), followed by Forth-79 in the early months of 1980.

Forth-79 became established as the "official" Forth and has been widely accepted and supported by the industry and technical publications.

Forth-83 is an extended and updated version of Forth-79 and is to all intents and purposes, a superset of it. Forth-83 has been adopted by the American National Standards Institute as A.N.S.I. Standard Forth.

0.4 IS-FORTH.

IS-FORTH conforms to Forth-83 and includes the optional double-precision word-set and an assembler vocabulary. It is considered to be a particularly fast implementation and was written by Intelligent Software's programming team – who also wrote the IS-BASIC, EXOS operating system and WP word-processor implemented on the Enterprise-64 and 128 Computers.

As implemented on the Enterprise, it additionally includes over a hundred words, specifically added, to make the most of the unique features of the hardware and operating system.

Whenever a new word was required, that had not been defined in the Standard, attempts were made to take words from other existing Forths, where appropriate, rather than to use an arbitary choice. Users familiar with IS-BASIC will recognise, and soon be using, many of their favourite commands and variables – in Forth form!

0.5 FORTH and This Manual.

The central concept of FORTH is simply that one is provided with a certain number of basic commands – called words. These words may be built up into more powerful ones, by combining them. These, in turn, can be built up until, finally, one is left with one "super" word, which, when executed, will perform the desired function. This would be called a "program" in BASIC, but is called an "Application" in FORTH.

We will look at the basic operations, in FORTH, showing how words may be combined using the "colon definition" and how FORTH is used, in immediate mode. We will examine FORTH's mathematics, decision-making structures, strings etc. Then it will be seen how these are all brought together when applications are written into BLOCKs, for convenient storage, editing etc.

Loading FORTH From Tape.

* Insert the BASIC cartridge, connect and power-up in the usual way.
* Press the START key (FUNCTION KEY 1) and load the FORTH tape as if it were a BASIC program.
* If you have already been using BASIC, type LOAD or LOAD FORTH

The IS-FORTH banner will be shown and the amount of available memory displayed. The message "OK" and the flashing cursor assures us that all is well and that Forth has finished executing any commands given to it and is ready to accept a new one.

Note the "CAPS" message displayed at the top-left of the screen. This was chosen as the default condition because Forth only recognises upper-case characters in program commands etc.

The 'STOP' and 'HOLD' keys and the console joystick still function as described in the Enterprise Manual.

CHAPTER 1 – The Forth Bridge.

1.0 The purpose of this section.

This section is designed to help experienced BASIC programmers "boot-strap" themselves into FORTH programming (Perhaps it should be called "A Dabbler's to Guide Forth" !). This shows some of the similarities and differences between the two languages. NOVICES SHOULD SKIP THIS CHAPTER .

Having loaded FORTH (Instructions on how to load FORTH will be found in Chapter 2), type VLIST. This displays all the words that FORTH understands at the moment. This list can be added to.

1.1 THE STACK.

All operations, in FORTH, use the stack. It's operation is described in full, later. Items are pushed onto the top of, and removed from the top of, the stack. Typing .S will display the contents of the stack. When a number is pulled from the stack and used, it is destroyed and cannot be reused.

1.2 PRINTING & SIMPLE FUNCTIONS.

Printing in Forth uses the "." operator called "print". It differs from the BASIC PRINT statement in that it has no inherent carriage return (should this be needed then adding a space and "CR" will remedy this). The top item, on the stack, is displayed and destroyed. Reverse Polish Notation (used throughout Forth) means that the order in which things are entered, is different. IE.

```
BASIC                              FORTH
-----                              -----

PRINT 5                               5 . CR
PRINT 5*6                           5 6 * . CR
PRINT 12/3                          12 3 / . CR
PRINT"Hello!  I'm  Nick Chip"    .(  Hello!  I'm  Nick Chip) CR
```

NB. The space after ' .(' is necessary and will not be printed.

1.3 VARIABLES AND CONSTANTS.

In Forth one needs to differentiate between constants and variables. When a constant's name is used, the value that it represents, is pushed onto the stack for use. A variable's name, however, represents the address, in memory, where the value is stored – hence the difference in usage.

1.3.1 Variables.

In FORTH, variables need to be declared before use – normally at the beginning of the program (called an application in FORTH). A simple variable may be declared as follows :-

```
VARIABLE FRED
```

Trying to use an undeclared variable will result in an error message. To store a value in a variable we use the operator "!" called "store". To retrieve it, we use the operator "@" called "fetch". Assuming that we have already declared the variable :-

```
         BASIC                          FORTH
         -----                          -----
      LET FRED = 32                   32 FRED !
    LET FRED = 3 * 9                3 9 * FRED !
       PRINT FRED                     FRED @ . CR
  LET FRED = FRED / 2            FRED @ 2 / FRED !
```

Just entering FRED will place the memory address, that has been allocated in the dictionary on the stack. Note that if VLIST is executed (this lists all the words recognised by FORTH) an entry called "FRED" now exists.

1.3.2 Constants.

In BASIC, variables would be used (less efficiently), in the place of FORTH variables AND constants.

```
   BASIC                   FORTH
   -----                   -----
  LET X=56             56 CONSTANT X
  PRINT X                 X . CR
```

1.4 The Colon Definition.

We will only deal here with immediate mode operation. FORTH programs (or applications) are, in effect, new commands which are added to the language. New commands are defined as follows.
A colon is used to signal to FORTH that this is the start of a new definition. This is followed by the name of the new word, the operations to be carried out and is concluded with a semi-colon. EG.

```
: HELLO CR ." Hi there! I'm Nick Chip and pleased to meet you !" ;
```

Note that ." is used to print a string in a colon definition, whilst .((see above) is used in 'immediate mode'.

Typing in the above will result in a new dictionary entry called "HELLO". Whenever HELLO is typed in, the string enclosed by quotes will be displayed.
We may now use HELLO in further definitions. EG.

```
: HI HELLO CR ." Look at all the words that I know" VLIST ;
```

And so on

1.5 SOME SIMPLE PROGRAMS.

1.5.1 This Program Converts Farenheit to Celsius.

```
        BASIC                           FORTH
        -----                           -----

    10 INPUT A                    : DEGCON CR 32 - 5 * 9 / . ;
    20 LET B=A-32
    30 LET C=B*5/9
    40 PRINT C
    50 STOP
```

Use the Forth program as follows. To convert 144 degrees Farenheit to Celcius enter :-

144 DEGCON

The answer will be displayed.

1.5.2 Programme 2 – DO LOOPS.

This program demonstrates the relative speeds of FORTHand BASIC.

```
        BASIC                          FORTH
        -----                          -----

    10 FOR X=0 TO 9999             : TEST 9999 0 DO LOOP ;
    20 NEXT X
```

These two examples should be tried and timed. It will prove to be a most suprising exercise! The FORTH version shows an empty DO LOOP.

1.5.3 More DO LOOPS.

This very simple program shows how a DO LOOP is used.

```
        BASIC                        FORTH
        -----                        -----

    10 FOR X=1 TO 4              : TEST2 5 1 DO ." *" CR LOOP ;
    20 PRINT "*"
    30 NEXT X
    40 STOP
```

The above simply prints FOUR stars on separate lines. Note that, in FORTH, the DO LOOP needed to be loaded with one MORE than the required number of stars.

1.6 The IF THEN Conditional Structure.

There is a very subtle difference betweeen the BASIC and FORTH versions.

BASIC VERSION

This inputs a number and tests whether it is greater than 5. If it is, it displays "Greater than 5" and if not, it displays "5 or Less ".

```
100 INPUT X
110 IF X > 5 THEN GOTO 2000
120 PRINT "5 or less"
130 STOP
2000 PRINT "Greater than 5"
2010 STOP
```

FORTH VERSION

In this version, it must be noted that THEN represents the end of the IF function (in some Forths the term ENDIF is used).
The syntax is as follows :-

N1 N2 Operator IF (action if true) ELSE (action if false) THEN

Where N1 is the number to be evaluated, N2 is the number to be tested against and the operator is a test like > = etc. etc.
EG.

```
: SIZE 5 > IF ." Greater than 5" ELSE ." 5 or less" THEN CR ;
```

1.7 Going Forth.

Using the above examples and one's previous programming knowledge, it should be possible to write simple programs in Forth using additional words from the reference section. Using interpretive mode, as used in this chapter, programs cannot be saved . Only a very brief sketch of FORTH has been provided so far, but, we hope that this section has whetted the appetite
to learn more. If the above section was found to be confusing then please do not worry – we covered a great deal of ground here – All of the above is covered in greater detail as we work through the manual.

CHAPTER 2 – Setting Forth

2.0 Big Fleas Have Little Fleas

A FORTH computer may be likened to a new-born baby. An infant has certain in-born capabilities that do not have to be learned. It knows how to make basic sounds, kick, eat etc. etc. but it cannot break-dance, write a letter or read a novel. All of these things are learned by building up new functions from combinations of old ones.

All of the brilliance and sparkle of the finest dancer in the world started with a crawl. There was then a first, faltering footstep. Skills were built up, combined with other skills, to produce new ones.

These skills also needed to be learned in the right order. It is pointless trying to tango, if one cannot walk.

2.1 Little Fleas Have Smaller Fleas

The FORTH computer is similar to the above. When loaded into memory, FORTH has a number of basic, simple commands which may be combined to produce new functions. These are in turn built on to produce more complex ones, until, the desired result is obtained. I.E. The whole program will result in one final WORD which, when executed, will perform the entire function. The way that these words are combined, to produce new ones, is by the use of the COLON DEFINITION and is looked at later in this chapter. First, though, we will need to know a little more about some of the basic words and how they are used.

2.2 SLOW/FAST.

IS-FORTH supports two speeds of operation. Slow-mode supports comprehensive error-checking and reporting and is designed to be used during the development phase when 'bugs' are being sought and programmer/operator errors abound. When the program runs satisfactorily, the fast-mode may be used.

NB. THE DEFAULT CONDITION IS FAST.

To enter slow-mode type :-
SLOW 'ENTER'

To return to fast-mode type :-
FAST 'ENTER'

It is recommended that ALL development work is carried out in slow-mode. Unlike some other Forths, The 'STOP' key will always stop a Forth application, EVEN in fast-mode.

Enter slow mode, as above, so that any errors may be trapped in the following examples.

2.3 VLIST and the Vocabulary.

Type :-

```
                    VLIST 'ENTER'
```

Observe the result. (Note that it finished by printing 'OK' at the end to show that it is ready for a new command.) Displayed are the words that Forth understands and are called the 'vocabulary'. They are stored in the 'dictionary' in memory order. The most recently defined word is at the top of the list.

These words are the names of commands, variables, pointers and constants. As we create new words (from combinations of old words) to form our program (called an 'Application' in Forth parlance) they will be added to this list.

It is worthwhile to spend a few minutes looking at the displayed list. Some of these words will be very familar to the Enterprise user eg. YELLOW and some a little strange eg. ALLOT . These words and their functions are described, in detail, in the reference section to the rear of this manual. Note that they are all UPPER – CASE ! The joystick may be used so that the list may be more easily studied.

Do NOT attempt to use any of these words without understanding their functions as it will almost certainly result in a confused computer (and programmer!).

2.4 Forth Form – Reverse Polish Notation.

The thought of Reverse Polish Notation (or RPN as it is known), for some reason, seems to strike terror into the hearts of some would-be Forth programmers. RPN is a little unusual to look at and may take a little time to master, but it is nothing to be afraid of. If you have used a Hewlett-Packard programmable calculater, RPN will already be familiar. However, if you were bad at algebra at school, RPN could be just what you are looking for (It is not algebraic !).

2.4.1. Addition.

Try this, making sure that the spaces are included. This is the equivalent of five plus four.
5 4 + 'ENTER'
By now, you may be wondering why have no answer – only an 'OK' and flashing cursor. Forth has actually completed the calculation but has left the answer on the stack (more about this later). To print the answer type a full stop and depress the 'ENTER' key The screen should show :-
5 4 + OK
NB. that the '.' does not produce a new line or return the cursor to the left margin. It merely prints the top item from the stack and is called 'print'.

Other operations are handled similarly. Try the following, remembering to press the 'ENTER' key to obtain the result :-

2.4.2 Subtraction.

```
12 4 - .
```

2.4.3 Multiplication.

```
12 2 * .
```

2.4.4 Division.

```
12 3 / .
```

So, all we needed to do was to enter the two numbers (don't forget the spaces!) and then the operator. This left the result on the stack for us to print or to use in later calculations eg. :-

```
12 4 + 2 / .
```

This will give the answer 8 (because 12 plus 4 equals 16 (left on the stack) and 16 divided by 2 is 8). EVALUATION OF AN EXPRESSION IS ALWAYS FROM LEFT TO RIGHT which means that there is NO OPERATION PRECEDENCE IN FORTH. In the above example the addition was carried out before the division – quite correctly from a RPN standpoint.

Compare these examples :-

FORTH	BASIC
5 3 + .	PRINT 5+3
12 3 / .	PRINT 12/3
3 4 * .	PRINT 3*4
12 9 - .	PRINT 12-9
10 2 + 2 * .	PRINT (10+2)*2
11 3 * 4 - .	PRINT 11*3-4
12 2 + 7 / .	PRINT (12+2)/7
2 4 * 3 + .	PRINT 3+2*4

10 2 2 + * .   PRINT (2+2) * 10

The Forth and Basic forms shown are mathematically identical, but are presented differently. Thus, formulae must be translated into RPN with care, otherwise the wrong answer will result. After a short while of using Forth, one tends to start thinking in RPN (!) and all becomes a little clearer. It is advised that the above examples should be studied until an understanding is obtained as EVERYTHING in Forth is based on RPN – including ALL Forth words.

2.5 FORTH, Numbers and Arithmetic.

Forth does ALL of it's arithmetic in binary. It converts the input numbers to binary – performs operation upon the data – and then reconverts from binary to the decimal (or whatever) when the result is printed.

2.5.1 Signed Single-Precision Numbers.

In Forth, numbers are normally stored and handled as 16 bit signed binary numbers (single-precision). This means that numbers in the range of -32,768 to +32,767 may be represented. All the examples, so far, have used numbers of this type. The "." operator prints out the binary value, converted to the number base that we are using, in this form.

2.5.2 Unsigned Single-Precision Numbers.

If only positive numbers are being used, it is possible to use numbers in the range 0 to 65,535 (unsigned single- precision). However, to print out numbers in this form we must use the unsigned print operative :-

U.

2.5.3 Double-Precision Numbers.

When larger numbers are required, double-precision may be employed. These are stored as 32 bit quantities and may be in the range -2,147,483,648 to +2,147,483,647 for signed numbers, or 0 to 4,294,967,295 for unsigned. Most double-precision operations are prefixed by a "D" . Here are some double-precision equivalents to some commonly-used functions. Further details and functions will be found in the reference section :-

```
D.     This is the double-precision version of     .
D+                   Ditto                         +
D-                   Ditto                         -
D*                   Ditto                         *
DU.                  Ditto                         U.
```

To push a double-precision number on to the stack, all one has to do is to add a decimal point to the end of the number. In fact, a decimal point,

anywhere in a number, signals to FORTH that it should be treated as a double-precision number.

Eg. Enter the following:-

```
1234. 2000. D+ D.
```

The result will be :-

```
3234
```

Using double-precision, when it is not necessary, will result in speed penalties. To print an unsigned double-precision number from the stack use:-

```
DU.
```

2.5.4 To Convert 16 bit Number To A 32 bit Number.

S->D

This sign-extends a single-precision number to a double precis-ion one. When working in mixed precision this is often required.
EG.

```
1234 S->D D. 1234 ok
```

2.6 Number Bases.

Working to different bases is very easy. FORTH has a variable called "BASE". On power-up this is set to 10 (that isten in baseten) ie. decimal operation. It converts the input (which it assumes to be in the base stored in BASE) into binary, for storage and use. Output is the reverse of this process.

DECIMAL

This word sets the machine to decimal working and is the default condition. Loads BASE with decimal 10.

HEX

This sets the machine to base 16 operation. It is very useful for use with the resident Forth Assembler and for machine-code operations as one byte (8 bits) are represented by only two digits. BASE is loaded with decimal 16.

OCTAL

Base 8 operation. Unlikely to be useful in everyday operations. It was widely used on 12 bit computers (The Enterprise is an 8 bit machine). BASE is loaded with decimal 8.

BINARY

Binary working. This can be very useful for such things as entering video characters, control and displaying the state of the machine, ports etc. BASE is loaded with decimal 2

Setting BASE , directly, is also possible Eg. :-

```
DECIMAL 7 BASE !
```

will set the machine to base 7 working. The "!" operator is called "store" and stored decimal 7 in variable BASE

A seemingly unlikely base to work in is Base 36 ! The usefulness of this base is that all the alphanumeric characters can be represented as numbers and so characters may be sorted, words may be treated as numbers for word-matching etc. etc. using arithmetical operators. See the Character strings section for further details.

Example. To convert 1234 Decimal to Hexa-decimal.

Type in the following, followed by "enter" :-

```
DECIMAL 1234 HEX . DECIMAL
```

The answer is " 4D2 "

Note the "DECIMAL" at the begining – this is to make sure that we are in base 10 before entering the data. Returning the machine to decimal avoids possible confusion later.

Example. To convert a Binary Number to Decimal.

The following may be entered :-

```
BINARY 010001001010 DECIMAL .
```

The answer is 1098. Try substituting other binary values.

Example. Base 36 Working.

Try the following (not too serious) entry :-

```
BOY + GIRL + MOON
```

Convert to RPN

```
36 BASE ! BOY GIRL MOON + + . DECIMAL
```

The answer is 78Q. (Was Shakespeare wasting his time?)

2.7 The Colon Definition.

This how we extend the power of FORTH. When Forth encounters a colon (:) it takes it as a signal to create a new dictionary entry in the name of the word which follows. It also sets the machine into "compile" mode so that the input stream, that follows, is compiled into the new dictionary entry. When a semi-colon is encountered, the compilation is completed.
EG.
There is no direct equivalent in FORTH of the BASIC PRINT ie. there is no carriage return in the FORTH version. So, let us create one.

```
: PRINT . CR ;
```

When the above is entered, a new dictionary entry called "PRINT" will be created (Typing VLIST will show this). Now when we type PRINT the top stack item will be printed, followed by a carriage return/linefeed – as in the BASIC version. We can now use this word in new definitions.

```
: BIN BINARY PRINT DECIMAL ;
```

The function of this new word is to set the machine to binary operation, take the top stack item and display it in binary form and then return the machine to decimal operation. So, if we place a decimal number on the stack and then type BIN it will diplay it in binary form.
EG.

```
10 BIN
```

This will displav 1010

We could use now BIN in further definitions as it is now just another FORTH word. Let us write a word that is the hexadecimal form of BIN.
EG.

```
: HX HEX PRINT DECIMAL ;
```

HX will take the top stack item and display it in hexadecimal form

```
: SHOW DUP CR BIN SPACE HX CR ;
```

(Two new words here. DUP duplicates the top stack item and SPACE prints a space).

So, SHOW will take the top stack item and duplicate it. It will then print out the binary equivalent, followed by a space, and then will use the duplicated stack item to print the hexadecimal form and finally moves the cursor to the start of a new line, for neatness.

CHAPTER 3 – The Stack.

3.0 The Stack.

Actually, there are two stacks in a Forth machine – the Parameter Stack (commonly just called 'the Stack') and the Return Stack (which Forth uses to thread it's way through the application and for other house-keeping purposes and may be used, with care, by the user in certain circumtances - see later).

The stack is a temporary storage device used by Forth to hold numbers, before processing, and to store the result. It is also used for passing numbers and information from one part of an application to another and for the temporary storage of data.

One can imagine the stack as being a pile of cards with numbers written upon them. To start with (eg.from power-up) the pile is empty and so nothing can be removed from it. Cards are placed on, and removed from, the top of the pile producing a, so called, last-in first-out storage system. Let us examine the Stack in greater detail.

3.1 Stack-Print .S

The '.S' operator prints the contents of the stack, without destroying the contents and can be very useful during program development.

Type this :-

```
.S
```

The machine should respond with 'OK' indicating that the stack is empty. (If anything else is printed, do not worry, simply type SP! which clears the stack and repeat the above.). Now type the following (remembering to press ENTER afterwards) which will load these numbers on to the stack :-

```
1 2 3 4 5
```

The machine will respond with 'OK.' Now enter '.S' and the stack contents will be displayed with the top-most item on the extreme right.

Now enter :-

```
.
```

The '.' prints AND REMOVES the top item on the stack. Enter:-

```
.S
```

The result will be -

```
1 2 3 4 OK
```

3.2 Stack Arithmetic.

Let us now see what happens during Forth addition. We already have the above four items on the stack so let us now enter :-

```
+ .S
```

The result will be :-

```
1 2 7 OK
```

This is because the '+' operator takes the top two items from the stack, adds them together and places the result back on the stack ie. 7 . NB. It does not matter how deep the stack is the '+' always takes the top two items to work on. Subtraction, multiplication and division work similarly.
Try this :-

```
* + .
```

The answer is 15 which will have been printed. This is because Forth evaluates from left to right – ie. 7 multiplied by 2 leaves 14 on the stack. 14 plus 1 leaves 15 on the stack which is then printed by the '.' function.

Typing '.S' will reveal that the stack is now empty. Try a further '.' and observe the result (Any attempt to remove an item from an empty stack will result in a 'STACK UNDERFLOW' error message).

3.3 Stack Manipulation.

When performing arithmetic, despite careful planning, it is often found that the stack data is in the wrong order for convenient processing. It may be rearranged using the stack operators.

DROP

After calculations, it often occurs that miscellaneous data is left on the stack which MUST be removed. "Clean-up as you go" is as good a motto in Forth as it is in the kitchen. DROP removes the top stack item and does nothing else.

2DROP

This drops the top 32 bit stack item (it will also remove two ordinary items).

DUP

Forth destroys stack data as it is used and data is often required several times in a calculation. This operator copies the top stack item. For example :-

```
                              3 3 * .
is the same as :-
                              3 DUP * .
```

2DUP

This duplicates the top 32 bit stack item (it will also duplicate two ordinary items).

?DUP

This duplicates the top item if it is not Zero.

SWAP

If we were adding two numbers together, it would not matter in which order the top two items appeared, but, if we were dividing, it would be a different situation (eg.12/3 is not the same as 3/12). SWAP exchanges the top two items to help solve problems like this.

```
Eg.
                              1 2 3 4 5 SWAP
would produce :-
                              1 2 3 5 4
```

2SWAP

Swaps two 32 bit items.

OVER

This copies the second item on the stack, on to the top, without destroying the original item.

```
Eg. With the stack containing:-
                              1 2 3 4 5
Entering:-
                              OVER
would produce :-
                              1 2 3 4 5 4
```

2OVER

32 bit equivalent of OVER.

ROT

The top three items on the stack are rotated to the left by one position, thus bringing the third item to the top.
Eg. If the stack contains :-

```
                    1 2 3 4 5
```

Then entering :-

```
                      ROT
```

will result in

```
                    1 2 4 5 3
```

2ROT

This rotates the top three 32 bit items.

-ROT

This is the same as ROT but rotates in the OTHER direction.
Eg. If the stack contained :-

```
                    1 2 3 4 5
```

Entering :-

```
                     -ROT
```

would produce

```
                    1 2 5 3 4
```

-2ROT

32 bit equivalent of -ROT .

PICK

This copies an item from anywhere in the stack to the top. It is used as follows:

0 represents the top stack item and 1 represents the next item etc. So if we wanted to PICK the fourth item in the stack and the stacks contents were:-

```
                    1 2 3 4 5
```

we could enter:-

```
                     3 PICK
```

The stack would now contain:-

```
                   1 2 3 4 5 2
```

If we had wanted the second item down we would have entered "2 PICK" etc. It is interesting to note that "0 PICK" is equivalent to "DUP" and "1 PICK" is equivalent to OVER .

ROLL

This operator rotates a item from the stack to the top. This is similar to ROT but the rotation may be any number of items (ROT rotates three items). Eg. Original stack contents:-

```
                         1 2 3 4 5 6 7
```

Entering :-

```
                           4 ROLL
```

would produce

```
                         1 2 4 5 6 7 3
```

2 ROLL is equivalent to ROT

3.4 Stack Depth.

It is very unlikely that the stack will fill the system memory, in any correctly running application. However, if a loop, that has been created, leaves data on the stack, then a stack over-flow may occur. To guard against such eventualities, checking the stack depth during trial application executions is advisable. The operator DEPTH can be useful for this. When executed, the stack depth BEFORE the execution of the word is left on the stack and so may be printed using "." It may also be used in control loops. Eg. If the stack depth exceeded, say 2000 (Most unlikely in most normal applications) then some corrective action could be initiated and a program crash avoided. This could be implemented very simply with an "if greater than" statement (see later). Good programming practice will avoid this sort of problem.

CHAPTER 4 – Constants & Variables.

4.1 The FORTH Constant.

Forth has two sorts of constants viz. 16 bit and 32 bit. The most commonly used is the single-precision (16 bit) form. Constants must be declared before use. They are defined in the form :-

```
                    (quantity) CONSTANT (name)
EG.:-
                         36 CONSTANT YARDS
```

Once this has been entered in the dictionary (eg. by typing in the above etc.), then whenever YARDS executed, 36 will be placed on to the stack for future use.

4.1.1 Example :-

This simple program converts currency. Recently, there were 240 Yugoslavian Dinars to the English Pound (#). To convert from Pounds to Dinars all we have to do is to multiply the number of pounds that we have by the exchange-rate ie. 240. The exchange-rate is a constant and will not change during the program execution.

Our application will look like this :-

```
240 CONSTANT DINARS
: POUNDS CR DINARS * . ." DINARS" CR ;
```

So to change 15 Pounds to Dinars we would enter :-

```
15 POUNDS
```

and the computer would respond with :-

```
3600 DINARS
```

If we had a very large application, which used the exchange-rate in many places, then all we would have to do when the rate changed, would be to alter the declaration.

4.2 Double-Precision Constants.

These are used as above as follows :-

```
234000. 2CONSTANT BIGNUMBER
```

Thus, executing BIGNUMBER will place 234000 on the stack in double precision form. The full-stop (period) after the number, in the declaration, indicates to FORTH that this is a double-precision number.

4.3 The FORTH Variable.

The FORTH variable exists in two forms – single-precision and double-precision. All variables MUST be pre-declared before use.
To declare a single precision variable called TOTAL enter this :-

```
VARIABLE TOTAL
```

4.3.1 Storing Variables.

Variables cannot be used until they have been declared and a value stored in them. This may be done as follows :-

```
15 TOTAL !
```

When the word TOTAL is executed it actually places it's address on the stack. The "!" (called "store") stores the third stack item (in this case 15) in the memory address pointed to by top stack item (which TOTAL obligingly placed there)
A convenient way of setting a variable to zero is provided by 0! This is used :-

```
TOTAL 0!
```

4.3.2 Displaying Variables.

? Called "question-mark". To display the contents of a variable all one needs to do is then following :-

```
TOTAL ?
```

The contents of TOTAL will be displayed. The contents are unaffected. "?" cannot be used in calculations, except to display results as it leaves the stack unchanged.

4.3.3 Fetching Variables.

@ Called "fetch". Before a variable is used, in a calculation etc. , it must be first transfered to the stack. Executing the variable name will place the variable's address on to the stack. The "fetch" operator uses this to bring a copy of the variable's contents to the top of the stack.
EG.

```
28 TOTAL ! (sets TOTAL to 28)
TOTAL @ (fetches contents)
. (displays stack)
```

This would display 28 ie. the contents of TOTAL.

4.4 Handling Variables.

There are several special variable-handling words that are very useful.

4.4.1 Incrementing Variables.

1+! Called "one plus store", this is a frequently needed function for moving graphics, scores etc. and adds one to the contents of the named variable.

```
TOTAL 1+!
```

4.4.2 Decrementing Variables.

1-! Called "one-minus-store", this is also extensively used. It reduces the contents of the named variable by one.

```
TOTAL 1-!
```

4.4.3 Adding To Variables.

+! Called "plus-store". Very often we need to add numbers to a running total. This is designed for that function. To add 23 to variable TOTAL :-

```
23 TOTAL +!
```

4.4.4 Subtracting From Variables.

-! Called "minus-store". This subtracts the top stack item from a variable. For instance, to remove 12 from the contents of TOTAL we would :-

```
12 TOTAL -!
```

4.5 Double-Precision Variables.

These are used similarly to the above. This would declare a 32 bit variable called "ROBERT" (The operator is called "two-variable") :-

```
2VARIABLE ROBERT
```

To store a 32 bit stack item in it, we would use 2! (called "two-store").
EG.

```
21346. ROBERT 2!
```

To fetch it we would use 2@ (called "two-fetch"). The example below fetches and displays the contents of ROBERT.
EG.

```
ROBERT 2@ D.
```

4.6 8-Bit Handling of Variables.

The least significant 8 bits (lower byte) of a variable may be acessed using C@ (called "c-fetch") and C! (called "c-store").

The main advantage of this is speed. It takes less time to access 8 bits than 16 or 32, on an 8 bit bus. These are used like ! and @ .

CHAPTER 5 – Functions.

5.1 Further Arithmetical Functions.

Chapter 2 showed how FORTH deals with simple addition, subtraction, multiplication and division. Now let us look at some refinements.

5.1.1 Stack Arithmetic.

1+
–

Very often, we need to increment or decrement the stack by one or two. The obvious way to do this would be :-

```
1 +
```

This is relatively slow and so other operators are employed – in this case 1+

```
EG. 23 1+ . 24 ok
```

2+ is similar but increments the stack by two.
– EG.

```
23 2+ . 25 ok
```

1- decrements the stack contents by one and is used as above.
–

2- decrements the stack contents by two and is used as above.
–

5.1.2 Division.

FORTH, it will be remembered, has integer (whole number) arithmetic. Special care, therefore, needs to be taken with division. By way of example, seven divided by three would give the answer (quotient) – two. The answer is, of course, not accurate as the correct answer is two with a remainder of one.

/MOD

This solves this problem. This performs a division and also leaves the remainder on the stack (remember when examining the example that the "." operator prints the top item first so that the remainder and result are displayed in the reverse order to that which they are placed on the stack).
EG.

```
7 3 /MOD . . 2 1 ok
```

MOD

This performs a division but ONLY leaves the remainder on the stack.
EG.

```
7 3 MOD . 1 ok
```

5.1.3 Composite Functions.

Designed to speed-up and simplify arithmetical operations, these operators can be most useful.

*/

This is probably best shown by example :-

```
3 4 2 */ . 6 ok
```

The first item (3) is multiplied by the second item (4) to produce a 32 bit result. This, in turn is then divided by the third item. In this case, the answer is 6 as shown above. This is not only a combination of * and / as the 32 bit intermediate result avoids posible overflows.

*/MOD

Similar to above, it also leaves the remainder on the stack.
EG.

```
2 5 3 */MOD .. 3 1 ok
```

The 2 is multiplied by the 5 to produce 10 as an intermediate result (held in double-precision form). This is then divided by the 3 to produce the remainder (1). This is pushed onto the stack, as is the quotient, which is the top stack item.

UM*

This is an UNSIGNED multiply of two 16 bit numbers, to produce a 32 bit, unsigned, result.
EG.

```
1200 2000 UM* D. 2400000 ok
```

UM/MOD

This function divides a 32 bit number by a 16 bit number, leaves the remainder on the stack and the quotient as the top item.
EG.

```
1234567. 666 UM/MOD . . 1853 469 ok
```

5.2 Handling Signs.

ABS

Called "absolute" this function makes the sign of a 16 bit number positive regardless of it's previous sign.
EG.

```
-12 ABS . 12 ok
 12 ABS . 12 ok
```

DABS

Double-precision version of ABS
EG.

```
-1234567. DABS D. 1234567 ok
 1234567. DABS D. 1234567 ok
```

NEGATE

Changes the sign of a 16 bit number.
EG.

```
-34 NEGATE . 34 ok
 34 NEGATE . -34 ok
```

DNEGATE

Double-precision version of NEGATE.
EG.

```
-1234567. DNEGATE D. 1234567 ok
 1234567. DNEGATE D. -1234567 ok
```

5.3 Logical Functions.

It will be assumed, here, that simple logic is understood. These fuctions are explained, in detail, in any "introduction to computing" type of book. There is further information in the Enterprise Manual. These functions are all 16-bit bitwise operations. It is important to use the unsigned print operator U. to display the results of this kind of operation. Logical operations are essentially unsigned.

AND

The AND functions are very useful for isolating one bit of a status byte or port etc.
The truth table looks like this:-

A	B	Result.
0	0	0
0	1	0
1	0	0
1	1	1

NAND

Used similarly to the above this is AND function but with an inverted result.
Truth Table :-

A	B	Result.
0	0	1
0	1	1
1	0	1
1	1	0

OR

Useful for setting bits, in registers, without upsetting the other bits.
Truth Table :-

A	B	Result.
0	0	0
1	0	1
0	1	1
1	1	1

NOR

Used as above, this is the inverted output version of OR.

Truth Table :-

A	B	Result.
0	0	1
0	1	0
1	0	0
1	1	0

XOR

Exclusive-Or. This is used extensively for toggling bits and in graphics. If a bit is exclusively-or'd twice it reverts to its former state.

Truth Table :-

A	B	Result.
0	0	0
0	1	1
1	0	1
1	1	0

NOT

This function inverts all of the bits in the 16 bit number.

5.4 Arithmetic Shift Functions.

When performing very fast mathematical operations, eg. for graphics or control, the following operations can shave off precious micro-seconds. They are multiplication and division by arithmetically shifting bits to the left (multiply by 2) or the right (divide by 2). No remainders are left.

2/

Divide by two by arithmetic shift to the right. This is best illustrated in binary :-

```
BINARY 011000110 2/ . 01100011 ok
```

Note that the trailing bit has been lost.

D2/

Double-precision version of 2/ . Used as above in 32 bit situations.

2*

Multiply by two by arithmetic by arithmetic shift to the left. EG.

```
BINARY 0110001101 2* . 01100011010 ok
```

Note the extra digit (zero) that has been added to the right hand end of the number.

5.5 Minimum And Maximum.

This is similar to the MIN and MAX in IS-BASIC. Two numbers are placed on to the stack. They are evaluated and, in the case of MAX, the largest number is placed back onto the stack. In the case of MIN , the smaller of the two numbers is placed back onto the stack.

Examples :-

```
 12 3 MAX . 12 ok
 3 32 MAX . 32 ok
  12 3 MIN . 3 ok
54 62 MIN . 54 ok
```

DMAX and DMIN are the double-precision versions.

5.6 Random Number Generation.

Extensively used in games and demonstration programs this function is very easy to use. A random number may be generated in th range 0-65536.

To generate a random number between 0 and 100 and to leave it on the stack, we would use :-

```
101 RND
```

It will be seen that we need to load the stack with a number, one larger than, the highest number required.

Example – A Die Simulation.

Let us write a program to simulate a die and to introduce some new words. KEY halts the machine and waits for the any key depression before resuming program execution. The BEGIN..REPEAT loop is endless, unless the STOP-KEY is pressed.

```
: DIE BEGIN 6 RND 1+ . CR KEY REPEAT ;
```

When DIE is executed a random number between 0 and 5 , inclusive, is generated. We require a random number between 1 and 6 so we add one to the random number before displaying it. The CR moves the cursor down a line. The program operation is now suspended until any key is pressed. The program will then re-run until the STOP key is pressed.

CHAPTER 6 – Conditional Structures.

The power of a computer lies in its ability to evaluate, to make decisions and to act accordingly. FORTH has two powerful types of structures to facilitate this – the IF..ELSE..THEN and LOOP structures.

The IF...ELSE...THEN structure is similar in function to BASIC's IF...THEN...ELSE and is based on the premise that IF something is true, to take one course of action, ELSE (if not true) to take another course of action, THEN (end of function) carry on with the rest of the program.

LOOPS (in FORTH there are several varieties) continue taking one course of action until some condition is met (or fails to be met). The loop is then broken and the application progresses.

6.1 The Conditional Operators – True and False.

Both of the above types of structure set conditions. If these are met, then, the condition is held to be true. If not, the condition is said to be false. EG.

```
12 12 =
```

This is TRUE.

```
6 12 =
```

This is FALSE.

If the condition is true FORTH sets a FLAG, which is pushed onto the stack. In IS-FORTH, TRUE is represented by -1 and FALSE by 0
EG.

```
12 12 = . -1 ok

6 12 = . 0 ok
```

Whether the flag is set, or not, will determine future action in the loop or IF structure. Other operators are used similarly :-

Single Precision	Double Precision	Function.
=	D=	Equals
>	no equivalent	Greater-than
<	D<	Less-than
<>	no equivalent	Not-equal-to
U<	DU<	Unsigned-less-than

These operators compare the top stack item with zero, only.

0=	D0=	Equals-zero
0>	no equivalent	Greater-than-zero
0<	no equivalent	Less-than-zero

These are used like this:-

```
5 0> . -1 ok          (true)
5 0< . 0 ok           (false)
```

Two operators TRUE and FALSE are provided to aid programtesting and readability.
TRUE pushes -1 onto the stack.
FALSE pushes 0 onto the stack.

6.2 Simple IF..THEN

Using the operators that we have just learned in 6.1, we may now start to use the IF..THEN structure.
The Format is :-

(conditional test) IF (words for action if TRUE) THEN

Suppose that we wanted to see if the top stack item is a six and, if so, to print "Correct". We could write the following application :-

```
: SIX? 6 = IF ." Correct!" THEN ;
```

In this definition, the stack is evaluated, as in the last section, If the condition is TRUE then "Correct" will be displayed. If the condition is FALSE then no further action is taken. (The "THEN" signifies the end of the IF function only. It is NOT like the BASIC "THEN".)

SIX? would execute like this:-

```
7 SIX? ok
   6 SIX? Correct! ok
   5 SIX? ok
```

6.3 IF..ELSE..THEN

This is an extention of IF..THEN as described above. The format is as follows:-

(conditional test) IF (words for action if true)
ELSE (words for action if false) THEN

Example.

Suppose we wanted to extend the example in 6.2 to display "Correct!" if the top stack item is a six and to display "Incorrect!" if it is any other value. The modified application would look like this :-

```
: ?SIX 6 = IF ." Correct!" ELSE ." Incorrect!" THEN ;
```

When executed, it would look like this :-

```
5 ?SIX Incorrect!ok
   6 ?SIX Correct!ok
   7 ?SIX Incorrect!ok
```

6.4 Nested IF's

IF's may be nested without difficulty. Care must be taken to ensure that the number of IF's and THEN's balance, otherwise errors will occur. Because of the very nature of FORTH we may have many IF's nested within IF's without really realising it. This can arise if a word has an IF in it, which calls an earlier defined word, which also has an IF in it etc. etc. This should cause no real problems to a tidy programmer.

IF's may also be nested within the same definition, if required as may be seen from the next example.

Example.

Let us write a program to sort people into three categories, by age. Let us say that a child is someone under twelve, an adolescent is someone aged between twelve and eighteen and an adult is eighteen plus. Our sort program could look like this :-

```
: AGE DUP 12 < IF
                  ." Child"
              ELSE
                  18 < IF
                          ." Adolescent"
                       ELSE
                          ." Adult"
                       THEN
              THEN ;
```

Note how this has been formatted. All of the IF..ELSE..THEN 's that belong to each other (ie.the same "nest") are formatted vertically. Hopefully, this will help ensure correct structuring and so less "Bugs".

6.5 Loops.

There are several different sorts of loops, designed for different tasks. The BEGIN..UNTIL and BEGIN..WHILE..REPEAT are useful for repeatedly performing actions until something changes or alters, when another course of action is then embarked on. The DO..LOOP is useful for performing a task a certain number of times. There are also variations and additions to these.

6.6 DO..LOOP

6.6.1 Simple DO..LOOP

Instead of an infinite stream of stars, let us print twelve stars. To do this we will use the DO..LOOP which is similar, in function, to BASIC's For, Next Loop.

The format for this is :-

(Limit) (Index) DO (FORTH word) LOOP

Each time round the loop, the index is incremented until the limit value is reached. Thus the loop will be executed (limit-Index) times.

EG.

```
: STAR1 12 0 DO ." *" LOOP ;
```

Executing the above definition will print twelve stars. It is interesting to note that the following example will also perform the same function.

```
: STAR2 14 2 DO ." *" LOOP ;
```

DO..LOOPs may be nested.

6.6.2 The Index Value and the DO..LOOP

As we saw above, the index is incremented each time the loop is executed. It is often very useful to know the present index value and to be able to use this in calculations (similar to using BASIC's X in a "For X=1 to 8" type loop). The "I" and "J" operators are used for this.

I

Using "I" inside the DO..LOOP will place the index value, of the inner-most DO..LOOP, onto the stack.
EG.

```
: COUNTING 100 0 DO CR I . LOOP ;
```

Executing the above will display the numbers from zero to ninety-nine, on separate lines, using the index to generate the figures.

The index may also be used for calculations. For instance, this example will display the two-times table, using the index.

```
: TABLE 11 0 DO CR 2 DUP . ." TIMES" I DUP . ." =" * . LOOP ;
```

J

The "J" operator is used as above, but, holds the value of the next-innermost loop index

Example. To display the 0 – 10 times tables.

```
: INNER 11 0 CR DO CR J DUP . ." TIMES " I DUP . ." =" * . LOOP ;
: OUTER 10 0 CR DO INNER LOOP ;
```

The inner loop "INNER" is the same as TABLE except that instead of the figure "2", the letter "J" has been substituted. Thus, instead of the two-times table, INNER will display the table whose value is "J" (which is obtained from the outer loop). OUTER, of course, is there to increment "J" and then to call INNER.

6.6.3 DO...+LOOP

This is a variant of the DO..LOOP which enables the index (I) to be incremented by the value of the top stack item.
EG.

```
: THREES 101 0 DO CR I . 3 +LOOP ;
```

This will count from zero to 99 – in threes. This is only a very simple example and applications for compound interest tables etc. may easily be written using DO, I, J and +LOOP.

6.6.4 LEAVE

Sometimes we may come across a situation when we are in the middle of executing a large DO..LOOP when something happens to make us want to abandon it. This is what LEAVE is designed for.

EG. Let us modify the table displaying program, used above, but so that pushing any key will cause the loop to abort.

```
: INNER1 11 0 CR DO ?TERMINAL 0 <> IF
  LEAVE
  THEN
  CR J DUP . ." TIMES " I DUP . ." =" * . LOOP ;

: OUTER 10 0 CR DO INNER1 LOOP ;
```

?TERMINAL reads the terminal and, if there has been a key-press, its ASCII value is placed onto the stack. If there has been no key-press, then a value of zero is placed onto the stack. This is tested in the inner loop. If a key has been pressed ie. the stack is not equal to zero then the current loop in the currently executing word terminates.Otherwise the loop operation continues. To summarise, pressing a key causes the next table to be started at once.

6.7 BEGIN..UNTIL

This form of loop is used to sustain a course of action until a certain condition is met. This breaks the loop.

The format is :-

```
BEGIN (conditional test) UNTIL
```

For Example :-

```
: PAUSE BEGIN KEY 32 = UNTIL ;
: MONEY BEGIN ." $" PAUSE REPEAT ;
```

PAUSE is a loop which reads the keyboard until a "space" is input. MONEY prints a dollar sign and then calls PAUSE . When PAUSE passes control back to MONEY the action is repeated. The effect is that, if the space bar is held down, a stream of dollar signs is displayed. Any other key has no effect.

6.8 BEGIN..REPEAT

This is the simplest loop, in FORTH. The loop begins and continuously repeats itself until the stop key is pressed or the power supply is removed.

Example :-

```
                : STARS BEGIN ." *" REPEAT ;
```

This will print stars, endlessly.

6.9 BEGIN..WHILE..REPEAT

This is slightly more complex than the BEGIN..UNTIL loop. The general idea is that, while a condition is met, then the loop is sustained. As soon as the condition becomes false, then the loop is broken.

The format is :-

BEGIN (conditional test) WHILE (whatever words are required) REPEAT

For Example :-

```
        : DOLLARS BEGIN KEY 32 = WHILE ." $" REPEAT ;
```

This example will print dollar signs while the space bar is pressed. If any other key is pressed the loop is broken and FORTH reverts to command mode.

Further Notes.

Designing the optimum program structure is important for efficient programming and operation. Combining the "IF" type of structure with conditional loops can give tremendous power to the tidy programmer (and headaches and "bugs" to a sloppy one). Time spent in the planning stage will be well repaid later in the debugging phase.

CHAPTER 7 – Strings and Things

7.0 Forth and Strings.

Handling text, in FORTH, needs a little perseverance at first. For instance, there is no equivalent of INKEY$. Most text will be handled in string variables and string constants. Let us first look at these.

We have already used a string operator to output a single text eg.

```
.( STRING)
```

This, of course, is of limited use and we need some general way of storing and handling text.

7.1 The String Variable

It will be remembered from numeric variables, that executing the variable's name will place the variable's address on the stack, ready for other operators to work on. This is the same for a string variable, except that the first byte of the string contains the length of it. This means, of course, that the maximum length of a string is 255 characters (the maximum number that can be represented in 8 bits).

The string variable's name and its maximum length needs to be declared so that FORTH knows how much space to reserve.

EG.

```
80 $VARIABLE FRED$
```

This would declare a string variable with a maximum length of 80 bytes, called FRED$ .It is a good idea to add the dollar sign to the end of the variable (as above) to distinguish it from numeric variables. Executing FRED$ will place the address of the start of the string, on to the stack.

7.2 The String Constant

This is used mainly to conserve memory as it only stores what it actually uses. It is declared :-

```
" Hello" $CONSTANT GREETING$
```

This will store the string "HELLO" in the dictionary. When GREETING$ is executed, the address of the string is pushed onto the stack.

7.3 PAD

PAD is the address of a 255 byte general purpose storage area which is especially useful for string operations. The string handling words use PAD. It should only be used for temporary storage as defining new words etc. may overwrite it.

7.4 Outputting a String Variable.

To do this we use the word TYPE. It is used in the form

```
(address) (No. of bytes) TYPE

GREETING$ 5 TYPE
```

However, it will be remembered that the first byte of a string represents its length.

7.4.1 COUNT

So, what we need is an operator to take the byte count from the beginning of the string, to add one to the address, so that it now points to the beginning of the text and presents it in the form that TYPE expects. This is what COUNT does.
EG.:-

```
GREETING$ COUNT TYPE HELLOok
```

So, Let us define O$ which will output a string. We will be using this again.

```
: O$ COUNT TYPE ;
```

This is a most useful definition. To use it, just use the string variable (or constant's name) and then O$ as :-

```
GREETING$ O$ HELLOok
```

The above was to illustrate the use of COUNT and TYPE. FORTH, in fact, provides us with a string output operator which has the same action as O$. It is $. (called " STRING PRINT "). EG.

```
GREETING$ $. HELLOok
```

7.5 Inputting a String Variable.

There are many ways of doing this. This is one method.

7.5.1 EXPECT

This operator takes, from the input stream (Keyboard), a string which is terminated by a carriage return (ENTER), but is less than the number of characters stated, (Max. number 255) and stores the string at the address specified EG.

```
PAD 255 EXPECT
```

This will accept up to 255 characters, delimited by an ' ENTER ' and stores it in PAD. We cannot use this directly to store a character input string, in a string variable, because EXPECT does not supply the length byte.

7.5.2 SPAN

This, when used after EXPECT, places the address, of where FORTH holds the length of the string, onto the stack.

7.6 String Variable Input Example

Using EXPECT and SPAN we can now make a definition which will input a string, of up to 255 bytes, and store it in PAD, or any other predeclared string variable who's maximum length was set at 255. EG.

```
: $? DUP 1+ 255 EXPECT SPAN C@ SWAP C! ;
```

This will input the string, head it with the length byte and store it at the address, which has been previously left, on the stack.
EG.

```
PAD $?
```

The machine will now wait for the input string. After the string has been input type :-

```
PAD $.
```

and it will be displayed.

7.7 $! and $@ String-Store and String-Fetch.

These operators will fetch a string and place it in PAD, leaving it' address on the stack ($@) or store the string held in PAD ($!), in the stated string variable.

EG.

```
" FRED" $CONSTANT NAME$

NAME$ $@ $.
```

This declares and fetches a copy of NAME$ and places it in PAD. It then prints it.

7.8 Text Input From Within Defining-Words.

"

To initialise string variables and to input text into an application, this operator can be most useful. Its action is to take the string enclosed by the quotation marks and to place it into PAD. When used with $! it can be very helpful and also when passing parameters to EXOS. Note that there must be a space, which is not displayed, between the leading " and the string.

EG.

```
80 $VARIABLE JOE$
" I am Joe" JOE$ S!
```

7.9 CONCAT – Joining Strings Together.

Very often, one needs to join two strings together. This is called concatenation. To do this CONCAT joins the string, pointed to by the address on the top of the stack onto the end of the string in PAD, leaving the address of the combined string on the stack.

EG.

```
" -LOW" $CONSTANT LO$

" HIGH"          (puts "HIGH" into PAD)

LO$ CONCAT $. HIGH-LOWok
```

7.9.1 $<> String Comparisons.

$<>

This operator tests two strings to see if they are NOT equal. This is normally used as part of an IF ELSE THEN structure. Useful for checking keyboard entries against what was expected.

7.9.2 Example of the Use of String Variables.

```
" Hi! What is your first name ?" $CONSTANT H1$

" And your surname ?" $CONSTANT H2$

" I am glad to know you, " $CONSTANT REPLY$

" " $CONSTANT GAP$

20 $VARIABLE ISTNAME$

20 $VARIABLE SURNAME$

: $? DUP 1+ 255 EXPECT SPAN C@ SWAP C! ;

: HI CR H1$ $. CR ISTNAME$ $? CR H2$ $. CR

SURNAME$ $? ISTNAME$ $@ GAP$ CONCAT

SURNAME$ CONCAT CR REPLY$ $. PAD $. CR ;
```

The example executes like this :-

```
Hi! What is your first name ?

FRED                    (Key in reply)

And your surname ?

BLOGGS                  (   Ditto   )

I am glad to know you, FRED BLOGGS

ok
```

CHAPTER 8 – Blocks And Buffers.

8.0 General.

After looking at the principle features of FORTH, we are now ready to write some serious applications. However, using the direct entry colon definitions that we have used, so far, is time consuming and it is difficult to correct mistakes. Indeed, we have no record of the application, once it has been lost from the screen-buffer.

What we really require is some way of entering, storing, retrieving and editing applications, in an orderly fashion. FORTH does this by the system of blocks and buffers.

8.1 Blocks and Buffers.

Blocks (some Forth books refer to these as screens) are one kilobyte "chunks" of memory. These may be brought to the display for editing puposes. Applications are written into these blocks, which, when LOADed into memory, act as the input stream. ie. whatever has been typed into a block can be reloaded into the machine, just as if it had been entered from the keyboard, whenever required.

IS-FORTH does not expect blocks to be in the traditional FORTH format of 16 rows by 64 characters. Blocks may contain any number of lines, each with any number of characters. The total maximum size of any block is still 1K bytes, however, including a carriage-return/line-feed (Enter) at the end of every line. Therefore, either 40 or 80 characters per line formats may be used.

8.1.1 Blocks and Disk-Based Systems.

On a disk-based system, these blocks reside on the disk, are transferred into memory (the buffers) for editing and program creation and are then resaved back onto the disk. This is all transparent to the user.

When a block is accessed, using BLOCK, it may already be in memory from previous operations. In which case, the buffers allocated to the block in question will be displayed, ready for editing.

If the block required is not in memory, then, it will be loaded into an empty buffer, from the disk. Should all of the available buffers be full, then, the least-recently accessed block will be transferred back to the disk, before moving the required block into the vacated buffer.

8.1.2 Blocks and Cassette-Based Systems.

On a cassette-based system, the blocks can only reside in the buffers. Constantly moving blocks between tape and memory would be tedious and inconvenient. These buffers are allocated block numbers on an arbitary basis. Should there not be enough space for the application, an error will result. CREATE-BUFFERS will provide additional space (see ref. section).

8.2 FORTH Applications and Disks.

8.2.0 Saving Forth Programmes.

There are two ways of doing this using SAVE-BUFFERS and FLUSH-BUFFERS. EMPTY-BUFFERS has been included for completeness.

8.2.1 SAVE-BUFFERS

On a disk-based system, this will write out each block as a seperate file. The filenames will be derived from the block numbers. The data in the buffers will be unchanged.

8.2.2 FLUSH-BUFFERS

This performs the same operation as SAVE-BUFFERS but unassigns all of the buffers, ie, treating them as if they no longer contain meaningful information.

8.2.3 EMPTY-BUFFERS

This unassigns the buffers WITHOUT saving them.

8.3.1 Loading Forth Programmes.

This is done using LOAD-BUFFERS . It reloads the blocks, previously saved, using SAVE-BUFFERS.

8.4 Forth Applications And Cassette-Based Systems.

8.4.0 Tape Files And Names.

An application may be given a name by using the word NAME (details in the ref. sect.). After it has been named, any files produced will be called that name. If NAME has been executed, only files with the corresponding name will be loaded. This provides a method of isolating files on a tape.
Eg.

```
" PICTURE" $CONTANT TITLE$
TITLE$ NAME
```

This will name any subsequent tape files "PICTURE" and only files called "PICTURE" may be loaded.

8.4.1 Saving A Forth Programme.

SAVE-BUFFERS will save all of the current blocks onto tape.

8.4.2 Loading a Forth Programme.

LOAD-BUFFERS will load a tape that has been saved with SAVE-BUFFERS

8.5 Editing Blocks.

The FORTH word EDIT allows the editing of FORTH blocks. 20 lines of text are opened and displayed.
EG.

```
1 EDIT
```

This will open block one . The block may be entered or edited, in the normal way, using the Enterprise's editor as described in the Enterprise Manual. Examples of Forth programs, written in blocks are shown later.

Should one block be insufficient for the application, then, several may be linked together. The operator that performs this is:-

```
-->
```

Blocks, so linked, should be in consecutive, rising order. This should be placed at the end of the text, usually on the bottom line of a block. This command must not be used inside a definition. It will force the input stream to continue from the start of the next screen, regardless of whether any text or definitions appear after it in the screen.

8.6 Escaping From The Editor.

When the application has been entered, or the edit is complete, the ESCape key should be pressed. After a small pause, the display will revert to the normal text or graphics display. The entered text now resides in a buffer and may be recalled for further work by re-invoking the editor, as above.

The Editor's buffer, during this process, is normally 2K bytes in size, and so it is clearly possible to enter more than 1K bytes of text. Should this occur, then, when ESC is pressed the message:

Block too large: ABORT (Y/N)?

will appear on the status line.

The user now has four choices of reply :-

Y - The edit is aborted, and the contents of the editor's buffer are NOT copied to FORTH's block buffer.

N - The edit is not terminated,and can continue as though nothing had happened.

CR - (ENTER KEY) The edit is terminated and the contents of the editor's buffer are not copied to FORTH's block buffer. The editor's buffer remains intact, however, and another EDIT of the same block buffer will resume the edit.

ESC- The edit is terminated and only the first 1K bytes of text is copied into FORTH's block buffer. The editor's buffer remains intact.

When ESC is pressed (in either above situation), the block buffer is marked as updated, and so, on a disk system, may be written to disk, if another block is accessed.

In the above cases, where the editor's buffer remains intact, another edit of the same block will bypass the process of copying FORTH's block buffer to the editor's buffer.

NB. A few words (TEXT, LORES etc.) may close the editor and text channels used during the edit. Therefore the contents of the editor's buffer may be lost if these words are used.

8.7 Aborting An Edit.

To abort the editing process, the STOP key may be pressed.

The message:

STOP key pressed: ABORT (Y/N) ?

– is then displayed on the status line. The possible replies to this are as above, with the ESC key having the same effect as if it were pressed at the end of an edit.

The CR (Enter key) reply can be used to temporarily exit back to FORTH during an edit (eg. to try something out). This will not update FORTH's block buffer or lose the edited text.

8.8 LOADing Blocks.

When we have our neatly edited application in a block, we need to feed this into FORTH's input stream. This is done by using LOAD. The effect of this is just the same as typing in the whole of the block/blocks almost instantaneously! Any colon definitions will be compiled and any words outside of definitions will be executed immediately.

An application may be made self-starting by placing the start word at the end of the block. Thus, the application will compile and then execute when the application is loaded.

```
2 LOAD
```

This will load block 2 into the input stream.

```
2 5 THRU
```

This will load blocks 2 to 5, inclusive, into the input stream.

CHAPTER 9 – Using Enterprise's Special Features.

9.0 FORTH And EXOS.

The graphics, sound, cassette handling and channel control are just some of EXOS's features. EXOS forms the core of the machine's software and is called upon by the language, or other program that is in control. Many of the BASIC commands, that you have been using, are EXOS calls in disguise!
The Enterprise Technical Manual should be read, in conjuction with this chapter.
This version of FORTH provides additions to the vocabulary, so that these functions may be used with ease. These are Enterprise specific.

9.1 Using The EXOS Variables.

The words E! and E@ write to, and read from, EXOS variables respectively. They are similar to the standard words ! and @, but take the EXOS variable numbers instead of addresses.

The words ON and OFF write 0 or 0FFH to an EXOS variable, which often has the effect of turning a particular EXOS feature on or off (see Tech. Manual. The word TOGGLE will complement the contents of an EXOS variable.

Many EXOS variables are defined as FORTH constants, for ease of use, with one of the above words.

Some Examples Are:-

```
0  BORDER E!        ( either of these will  set
                      the border to Black.)
  BLACK BORDER E!

  STATUS  ON        ( will cause  the  status
                     line to be displayed.)
  REM1 TOGGLE
                    ( will toggle the state  of
                     Remote 1.)
```

9.1.2 The EXOS variables defined are:

BUFFER	TIMER
STOP	SPEAKER
BAUD (	FORMAT
MODE	COLOURS
COLORS	STATUS
X	Y
STATUS	BORDER
REM1	REM2
BIAS	

9.2 FORTH And Channels.

All input and output operations, on the Enterprise, are routed through the channel system. This system is described ,in detail, in the Enterprise Technical Manual and therefore we will only look at how FORTH interfaces with it.

Many FORTH words need to refer to a channel. Often several refer to the same channel. To save the user from having to give the channel number each time such a word is used, Enterprise FORTH has a small redirection table, from which channel numbers can be taken. Thus, to make PLOT, for example, go to a non-default graphics channel, the redirection table entry corresponding to the graphics channel is altered.

9.2.1 Changing Channels.

The words #EDITOR, #GRAPHICS, and #TEXT will set the default e d i t o r graphics and text channels to the preceding number.

For example :-

```
101 #GRAPHICS
```

– will restore the default graphics channel to 101.

The words ATTRIBUTE, HIRES, LORES and TEXT will restore the default values in the redirections table for the text and graphics channels. The initial state of the redirection table and the use of the channels are:

100	EDITOR:
101	VIDEO: (graphics)
102	VIDEO: (text)
103	SOUND:
104	PRINTER:
105	KEYBOARD:
106	TAPE: or DISK:
107	VIDEO: (during EDIT)
108	EDITOR: (during EDIT)

Note that certain words, such as CLOSE and GET do not go through the redirection table, but instead take the channel number as a parameter on the stack.

CHAPTER 10 – The Forth Assembler.

10.0 Intoduction.

This chapter is NOT an assembler tutorial and was never intended to be. It assumes a knowledge of Z80 Assembler and assembly techniques. There are many books on the subject, available from good book-shops and possibly the local library.

IS-FORTH includes a vocabulary containing a Z80 assembler, allowing the time-critical parts of an application to be made as fast as possible. Often a complete application will be developed in FORTH, then some central words converted to machine code using the assembler.

The assembler is not intended for writing complete application programs, but is useful for re-writing low-level FORTH words. It does not provide labels and is single-pass (being FORTH in style). However it is adequate for its intended purpose.

10.1 The Assembler Vocabulary.

The assembler vocabulary contains a number of FORTH words for compiling machine code into the dictionary, using the appropriate FORTH words (eg. CODE ... END-CODE).

All the assembler words, when executed, compile the machine code into the dictionary. Thus FORTH words, that introduce assembler, all turn off compilation. The words that end the assembler sequence – turn it back on again, if necessary.

If the assembler words are compiled into a definition, then they will, in turn, compile the machine code – when the definition is executed.

10.2 Assembler, FORTH and RPN.

As with the rest of FORTH, the assembler works in Reverse Polish Notation, with the operands coming before operators. Although this takes a little getting used to, it allows the normal power of FORTH to be used at assembly time, and allows the user's own assembler words to be defined, providing a powerful macro capability.

All Z80 registers are definied as constants, and the remaining assembler words are normal FORTH words which take their parameters from the stack, form the correct opcode, and then compile it into the next available dictionary location.

10.3 Z80 Registers.

All Z80 registers and condition codes are defined as 16-bit constants, the most significant byte defining the type of register (8-bit, 16-bit etc.) and the least significant byte defining the exact register within the type. The type byte (MSB) has values that are above C000H, thus allowing improved checking for stack errors, since programmers' addresses are unlikely to appear this high in memory.

Type bytes of 0FFH and 0 indicate an 8-bit number rather than a register.

The register constants and their values are (in hex.):

B	F000	(BC)	F800
C	F001	(DE)	F810
D	F002	I	F807
E	F003	R	F80F
H	F004		
L	F005	BC	E000
(HL)	F006	DE	E001
A	F007	HL	E002
		SP	E003
(IX+)	F606	AF	E003
(IX-)	F706	IX	EE02
(IY+)	F406	IY	EC02
(IY-)	F506		

The conditions to conditional jumps etc. are also defined as constants. The conditions NC, C, NZ, Z, PO, PE, P and M have the same values as the registers B, C, D, E, H, L, (HL) and A respectively. Thus the condition C and the register C, which have the same name, also have the same constant value:

NC	F000
C	F001
NZ	F002
Z	F003
PO	F004
PE	F005
P	F006
M	F007

The 'registers' (IX+), (IY-) etc. are equivalent to the Z80 (IX+d), (IY-d) etc. The displacement immediately precedes the register word eg. LD A,(IX-10) becomes 10 (IX-) A LD, LD (IX+10),A becomes A 10 (IX+) LD, and BIT 4,(IY-3) becomes 3 (IY-) 4 BIT,

10.4 The Instruction Set.

The instructions are similar to standard Zilog Z80 mnemonics, but adjust-ed to take account of the reverse polish notation used. The following is a list of Z80 mnemonics together with the FORTH equivalent:

Z80 mnemonic	*FORTH mnemonic*	*Z80 mnemonic*	*FORTH mnemonic*
ADC A,r	r A ADC,	IND	IND,
ADC A,n	n A ADC,	INDR	INDR,
ADC HL,rr	rr HL ADC,	INI	INI,
ADD A,r	r A ADD,	INIR	INIR,
ADD A,n	n A ADD,	JP (HL)	HL JP(),
ADD HL,rr	rr HL ADD,	JP (IX)	IX JP(),
ADD IX,rr	rr IX ADD,	JP (IY)	IY JP(),
ADD IY,rr	rr IY ADD,	JP nn	nn JP,
AND r	r AND,	JP cc,nn	nn cc ?JP,

AND n	n AND,	JR d	addr JR,
BIT b,r	r b BIT,	JR cc,d	addr cc ?JR,
CALL nn	nn CALL,	LD (BC),A	A (BC) LD,
CALL cc,nn	nn cc ?CALL,	LD (DE),A	A (DE) LD,
CCF	CCF,	LD R,A	A R LD,
CP r	r CP,	LD I,A	A I LD,
CP n	n CP,	LD A,(BC)	(BC) A LD,
CPD	CPD,	LD A,(DE)	(DE) A LD,
CPDR	CPDR,	LD A,R	R A LD,
CPI	CPI,	LD A,I	I A LD,
CPIR	CPIR,	LD r,r'	r' r LD,
CPL	CPL,	LD r,n	n r LD,
DAA	DAA,	LD (nn),A	A nn ()LD,
DEC r	r DEC,	LD (nn),HL	HL nn ()LD,
DEC rr	rr DEC,	LD (nn),DE	DE nn ()LD,
DI	DI,	LD (nn),BC	BC nn ()LD,
DJNZ d	Addr DJNZ,	LD (nn),SP	SP nn ()LD
EI	EI,	LD rr,nn	nn rr LD,
EX (SP),HL	HL SP ()EX,	LD rr,(nn)	nn rr LD(),
EX (SP),IX	IX SP ()EX,	LD SP,HL	HL SPLD,
EX (SP),IY	IY SP ()EX,	LD SP,IX	IX SPLD,
EX AF,AF'	AF AF EX,	LD SP,IY	IY SPLD,
EX DE,HL	HL DE EX,	LDD	LDD,
EXX	EXX,	LDDR	LDDR,
HALT	HALT,	LDI	LDI,
IM n	n IM,	LDIR	LDIR,
IN A,(C)	C A IN(),	NEG	NEG,
IN A,(n)	n A IN(),	NOP	NOP,
INC r	r INC,	OR r	r OR,
INC rr	rr INC,	OR n	n OR,

Z80 mnemonic FORTH mnemonic Z80 mnemonic FORTH mnemonic

OTDR	OTDR,	RR r	r RR,
OTIR	OTIR,	RRC r	r RRC,
OUTD	OUTD,	RRD	RRD,
OUTI	OUTI,	RST n	n RST,
POP rr	rr POP,	SBC A,r	r A SBC,
PUSH rr	rr PUSH,	SBC A,n	n A SBC,
RES b,r	r b RES,	SBC HL,rr	rr HL SBC,
RET	RET,	SCF	SCF,

RET cc	cc ?RET,	SET b,r	r b SET,
RETI	RETI,	SLA r	r SLA,
RETN	RETN,	SRA r	r SRA,
RL r	r RL,	SRL r	r SRL,
RLA	RLA,	SUB r	r SUB,
RLC r	r RLC,	SUB n	n SUB,
RLCA	RLCA,	XOR r	r XOR,
RLD	RLD,	XOR n	n XOR,

Note that all the FORTH assembler words end in a comma (,). This is for three reasons:

i) It is a FORTH convention that words, that compile into the dictionary, end in a comma (eg. , and C,).

ii) The comma distinguishes assembler-words from FORTH-words that would, otherwise, have the same name eg. XOR .

iii) Typical FORTH assembler code tends to contain more that one instructionper line (as opposed to conventional assembler code). Also,the register constants are usually put on the stack immediately before the assembler word to which they are the operands. The commas help to visually separate the separate instructions, in this case.

10.4.1 Jumps.

The jumps, shown above, all take an address on the stack as their destination.

ie. JR, ?JR, JP, ?JP, CALL, ?CALL, and DJNZ,

These are used for backwards jumps, and the word BEGIN, will put the current dictionary address on the stack, for a later backward jump. IE. BEGIN, is used at the destination of the jump or call.

For forward jumps,these similar words are provided :-

JR>, ?JR>, JP>, ?JP>, CALL>, ?CALL>, and DJNZ>, These compile their opcode and reserve one or two bytes for the displacement or address. They also leave the address of the reserved byte(s) on the stack.

The words END, and ENDR, are used, later at the destination of the jump, to fill in the displacement or address to the current dictionary address.
END, is used at the destination of JP>, ?JP>, CALL>, and ?CALL,

ENDR< is used at the destination of JR>, ?JR>, and DJNZ>.

10.5 Machine Code.

Through its built-in assembler, IS-FORTH provides easy use of machine code. Machine code sections are usually accessed in the form:

```
CODE <name> ... END-CODE
or : <name> ... ;CODE ... END-CODE
```

Certain Z80 registers in IS-FORTH are used for a consistant purpose throughout. These registers must be preserved or altered in accordance with the conventions.

10.5.1 The Registers.

The registers are used as follows:

SP - The Z80 stack pointer is used to point to the parameter stack. The top two items on this stack are held in registers (see below).

IY - Points to the least significant byte of the last 16-bit value item added to the return stack (ie. the same convention as the normal Z80 stack pointer SP).

IX - Points to the 'inner interpreter' called NEXT. This is entered every time a new word is called. Thus FORTH words coded in machine code must end in a IX JP() (ie. JP (IX))
instruction. NEXT is also a defined constant, so NEXT JP, can also be used to terminate a machine code section.

HL' - This is the 'interpreter pointer' and points to the next word to be executed within the colon definition currently being executed. When a new word is called, it is HL' that is pushed onto the return stack pointed to by IY, and then made to point to the new word.

DE'	- When a word is called, DE' points to the data immediately after the compilation address (CFA) in the definition. Thus if the FORTH word ." was written in machine code, then DE' would point to the string to be printed when the run-time code was executed.
HL,DE	- HL holds the top item on the parameter stack, and DE holds the second, and thus it is important that words written in machine code do not corrupt HL or DE, unless it is intend to alter the values on the stack.
AF'	- AF' is used to hold flags concerning the stop/break key, the background word, interrupt words (if available)and whether or not IS-FORTH is in the FAST mode. The actual use for AF' may vary from one implementation of IS-FORTH to another, so it should not be corrupted.

10.6 Speeding Up Forth Words.

The use of HL and DE to hold the top two items of the stack and the use IX to point to NEXT often speeds up FORTH words. For example, in IS-FORTH the code definition for SWAP is:

```
EX   DE,HL
JP   (IX)
```

In many Z80 FORTH implementations the equivalent to this would be:

```
POP  HL
EX   (SP),HL
PUSH HL
JP   NEXT
```

which is 50% longer and 24% slower.

Another example is the word 1+

In IS-FORTH the definition of this is:

```
INC  HL
JP   (IX)
```

A more conventional version would be:

```
POP  HL
INC  HL
PUSH HL
JP   NEXT
```

which is 50% longer and 38% slower.

It is recommended that machine code is not written initially. The whole application may be written all in FORTH and then the critical words converted when the algorithm has been proven.

CHAPTER 11 – Vocabularies.

11.0 General.

IS-FORTH supports the creation, and use, of separate sections of the dictionary, called vocabularies. Each vocabulary contains, typically a complete set of words for a particular application. This may, in turn, involve other vocabularies for separate sections of the application.

11.1 Vocabulary Structure.

Although each word is added to the top of the dictionary, providing a linear set of words in memory, the words are linked together such that a tree-structured dictionary results. Each branch of the tree corresponds to a vocabulary.

Each vocabulary can use whatever words it requires, without running the risk of re-defining a word that may be required, by some other application, or vocabulary, later compiled.

For Example :-

In the ASSEMBLER vocabulary, A is defined as a constant. If another application wanted to use A as a variable it could do so, providing it was the first vocabulary searched (ie. the CONTEXT vocabulary). If that application program wanted to include a machine code defined word, using the assembler, then CONTEXT would be set to the assembler vocabulary for assembling the code. It is then restored to it's original value, afterwards, for use as a variable.

11.2 Creating A New Vocabulary.

A new vocabulary is created by:

```
VOCABULARY <name>
```

When <name> is subsequently executed, it stores a pointer to itself in CONTEXT . This ensures that it is searched BEFORE the vocabulary, in which it is defined.

If new definitions are required to go into <name>, then the word:

```
DEFINITIONS
```

will put into CURRENT , the value that is in CONTEXT. CURRENT is the vocabulary into which new definitions of the vocabulary will be compiled.

When a vocabulary name is executed (<name> above), then each 'father' vocabulary will be made to continue the search at the end of it's father vocabulary (this process being recursive).

Thus, if a vocabulary called FRED is created, within the vocabulary FORTH, and new words are subsequently defined in FORTH, then words not found in the new vocabulary, FRED, will be searched for in the *whole* of vocabulary FORTH. It will not search, only, in the section defined before the definition of the new vocabulary, FRED.

For example:

```
FORTH DEFINITIONS  ( put new definitions in FORTH)
VOCABULARY V       ( create a new vocabulary)
V DEFINITIONS        ( put new definitions into V)
: T1 ... ;           ( new word in vocabulary V)
: T2 ... ;           ( another word in vocabulary V)
FORTH DEFINITIONS    ( back to FORTH)
: T3 ... ;           ( new word in vocabulary FORTH)
V                    ( back to V)
```

Even though the CONTEXT vocabulary is now V, T2 will still be found in a dictionary search, despite the fact that it was defined after the vocabulary V and all its definitions (T1).

The word VLIST will display all the words in the current search order, the first word displayed being the first word searched. The move from words in one vocabulary, to words in it's father vocabulary, can be seen in VLIST's display by an extra space between the bottom word of one vocabulary and the top word of the next.

In the example above, VLIST's display would be:

T2 T1   T3 V ... (rest of FORTH words).

Note the extra space between T1 and T3 where the vocabularies change.

CHAPTER 12 – Errors.

12.0 Errors – General.

The FORTH-83 Standard defines many 'error conditions'. A few of these (such as division by zero) IS-FORTH ignores. Others are normally given as a message to the user.

The error numbers and messages which may arise are:

1. <name> not found.
2. <name> control error.
3. <name> below FENCE.
4. Incomplete definition.
5. Stack overflow.
6. Stack underflow.
7. CURRENT not CONTEXT.
8. Can't FORGET CURRENT.
9. Buffers full.
10. Invalid condition. (assembler)
11. Invalid register. (assembler)
12. Invalid displacement. (assembler)

If an error number not covered by one of the above arrises, then the message:

Error type n.

– is given, where n is the error number.

12.1 Error Handling.

An error handler may be set active by:

(ABORT) <name>

where <name> is a previously defined word. It is then the responsibility of <name> to give error messages to the user. When (and if) <name>

finishes, the word ABORT will be executed. This resets the stacks and waits for more user input, without outputting any messages and without positioning the cursor.

When <name> is executed, the error number is on the stack, BLK has been copied to SCR and >IN has been copied to R#.

The word ERROR when used in the form:

```
n ERROR
```

(n is an error number), will cause an error as described above, except that any (ABORT) word defined will not be executed, allowing an (ABORT) word to give an error back to the user.

To disable the (ABORT) feature, the sequence:

```
(ABORT) ERROR
```

can be used.

CHAPTER 13 – FORTH Application Examples.

13.0 Introduction.

This section includes some example words written in IS-FORTH.

The first example is a simple memory dump word, which displays the contents of memory in hex. and ASCII.

The second example is slightly more complicated. It will show the exact location of errors that occur, whilst a block is loading. This can be very useful for debugging a large application.

The third example is more complicated still. It allows normal arithmetical expressions to by typed in. These are then evaluated correctly. Allowable operators are +, -, * and /. Parenthesis ('(' and ')') are also allowed.

The examples were edited into some unused blocks in the range 1 to 29, without the comments. The word INDEX can be used to see the contents of the blocks:

```
1 29 INDEX
( DUMP     :  displays 64 bytes of memory )
( PRIMARY  :  Reads in a constant and deals with paranthesis )
( TERM     :  Deals with * and / )
( EXPR     :  Deals with + and - )
( INFIX    :  Reads in an expression and evaluates it )
( .LINE    :  Prints from current position to end of line )
( WHERE    :  Shows where error during loading occured ) ok
```

13.1 Memory Dump.

This example gives a dump of memory, in hexadecimal, displaying 8 bytes per row, following the address. The ASCII representation of each byte is also displayed, if printable.

```
SCR # 1
( DUMP      :  displays 64 bytes of memory )
  : DUMP    ( addr1 -- addr2 )
      CR
      HEX
      8 0 DO                  ( print 8 rows )
        DUP                   ( addr )
        0 <# # # # #S #> TYPE 2 SPACES  ( display address )
        DUP 8 0 DO            ( print 8 bytes on the line )
          DUP                 ( addr )
          0 <# # # # #S #> TYPE SPACE ( display one byte )
        LOOP DROP
        8 0 DO          ( 8 ASCII characters )
          DUP C@ 127 AND      ( strip top bit )
          DUP 32 < IF         ( control character )
           DROP 32           ( replace with space )
          THEN EMIT 1+        ( display ASCII
        LOOP
        CR                    ( next row )
      LOOP
  ;
```

The word DUMP is used in the form:

```
        addr DUMP
```

Since it leaves the modified addr on the stack, it can be continued with another DUMP immediately it has finished.

For example:

```
HEX 1388 DUMP
1388  D1 73 23 72 23 22 04 30  Qs#r#" 0
1390  73 E1 D1 DD E9 06 53 4D  saQ]i SM
1398  ED 4B 28 30 0A EE 40 02  mK(0 n@
13A0  13 BB 13 ED 4B 10 30 78  ; mK 0x1
13A8  05 EB ED 5B 04 30 2B 72  km[ 0+r+
13B0  4E 54 49 4C B2 13 E0 13  NTIL2 '
13B8  13 7C B5 EB D1 C2 AD 02  |5kQB- Y
13C0  04 55 53 45 53 D6 13 01  USESV
ok
```

13.2 Error Locator.

Often, when loading from blocks, a compilation error occurs for some unknown reason. If the blocks being loaded all load together, it can be difficult to tell in which block the error occured. Even knowing this, it is useful to know exactly where, in the block, the error occured.

After an error, during loading, the FORTH error handler saves the value of BLK (the block number being loaded), in the variable SCR, and the value of >IN (the input stream offset) in the variable R#. It is, therefore, possible to write a FORTH word that will pin-point the erroneous word.

In the example below, the word WHERE, if executed immediately after an error during loading, will LIST the offending block, on the screen, and position an error pointer immediately below the erroneous word. The pointer used is ^***, which can easily be seen.

```
SCR # 20
( .LINE    :  Prints from current position to end of line )
  : .LINE   ( addr count -- addr count )
       BEGIN
         1-                    (decrement character count)
         SWAP DUP C@           (get current address and byte)
         SWAP 1+               (increment address)
         -ROT                  (addr 3rd, count 2nd, char top)
         DUP EMIT              (print character)
         10 =                  (repeat until ASCII line feed)
       UNTIL
  ;

-->

SCR # 21
( WHERE    :  Shows where error during loading occured )
  : WHERE
       CR
       R# @ IF                 (an error has occured)
         ." SCR # " SCR @ DUP .(print block number)
         CR                    (block number left on stack)
         BLOCK 1024 -TRAILING  (delete trailing spaces)
         OVER R# @ TYPE        (print block up to error)
         R# @ -                (adjust number of bytes left)
         SWAP R# @ + SWAP      (adjust address to after error)
         OUT @                 (get position of error on line)
         -ROT                  (get address and count on top)
         .LINE                 (print rest of erroneous line)
         ROT SPACES ." ^***" CR (print pointer on next line)
         TYPE                  (print rest of block)
       THEN
  ;
```

For an example of its use, suppose the example "DUMP" word, given above, was being loaded. If the first occurrence of LOOP had been misspelled LOP, then the message:

```
*** LOP not found.
```

would be given. Had WHERE been previously successfully loaded, then typing WHERE would produce:

```
WHERE
SCR # 1
( DUMP     :  displays 64 bytes of memory )
  : DUMP   ( addr1 -- addr2 )
      CR
      HEX
      8 0 DO                  ( print 8 rows )
        DUP                   ( addr )
        0 <# # # # #S #> TYPE 2 SPACES  ( display address )
        DUP 8 0 DO            ( print 8 bytes on the line )
          DUP                 ( addr )
          C@ 0 <# # #S #> TYPE SPACE 1+ ( display one byte )
        LOP DROP
           ^***
        SPACE 8 0 DO          ( 8 ASCII characters )
          DUP C@ 127 AND      ( strip top bit )
          DUP 32 < IF         ( control character )
            DROP 32           ( replace with space )
          THEN EMIT 1+        ( display ASCII
        LOOP
        CR                    ( next row )
      LOOP
  ;
```

The pointer ^*** points to immediately after the source of the error LOP.

13.3 Expression Evaluator.

This example consists of a series of words which will evaluate an infix (ie. conventional) expression, typed in, and will then print the result.

The word INFIX, when executed, will prompt for an expression to be typed. This expression can contain signed 16-bit integers, the operators +, -, * and / and parenthesis. No spaces should be included. / and * have a higher precedence than + and – so these operations are performed first, unless overridden with (and).

After evaluating one expression, INFIX will prompt for another. Pressing <return> alone will exit back to FORTH.

This example illustrates the ability of IS-FORTH words to forward reference other words. The expression is evaluated using a simple recursive, decent algorithm. The word PRIMARY, if it finds a left parenthesis ('(') , calls the word EXPR, to evaluate the expression within the parenthesis. EXPR however has to call TERM, which in turn needs to call PRIMARY.

Throughout the routines, the address of the current position, within the ASCII expression, is kept on the stack (addr) and each routine updates addr to point to, beyond, the item that it has just read.

For example:

```
INFIX
Expression: 2+3 =5
Expression: 2+3*4 =14
Expression: 2+(3+(4+(5*6)*4)*3)*2 =752
Expression: ok
```

```
SCR # 10
( PRIMARY  :  Reads in a constant and deals with parenthesis )
  : PRIMARY                    ( addr1 -- n addr2 )
    DUP C@ ASCII ( =
    IF                         ( open parenthesis ?)
      1+ FORWARD 1+             ( forward reference to EXPR )
    ELSE
      1- 0. ROT
      CONVERT SWAP DROP         ( convert ASCII constant )
    THEN
  ;
->
```

```
SCR # 11
( TERM     :  Deals with * and / )
  : TERM                      ( addr1 -- n addr2 )
     PRIMARY                  ( get number )
     BEGIN
       DUP C@ DUP ASCII * =   ( *  ?)
       SWAP ASCII / =         ( /  ?)
       OVER OR
     WHILE                    ( while * or / )
       SWAP 1+ PRIMARY ROT    ( get multiplicand or divisor)
       IF                     ( * )
         ROT ROT *
       ELSE                   ( / )
         ROT ROT /
       THEN
       SWAP
     REPEAT
     DROP
  ;

-->

SCR # 12
( EXPR     :  Deals with + and - )
  : EXPR                      ( addr1 -- n addr2 )
     TERM                     ( get number )
     BEGIN
       DUP C@ DUP ASCII + =   ( +  ?)
       SWAP ASCII - =         ( -  ?)
       OVER OR
     WHILE                    ( while + or - )
       SWAP 1+ TERM ROT       ( get other operand )
       IF                     ( + )
         ROT ROT +
       ELSE                   ( - )
         ROT ROT -
       THEN
       SWAP
     REPEAT
     DROP
  ;
```

```
PRIMARY USES EXPR              ( resolve forward reference )

-->


SCR # 13
( INFIX    :  Reads in an expression and evaluates it )
  : INFIX
     BEGIN
       CR ." Expression: " PAD 80 EXPECT
       SPAN @ ?DUP
     WHILE                       ( while an input was given )
       PAD + 0!                  ( terminate input with 0 )
       PAD EXPR ." =" DROP .     ( evaluate input, print result )
     REPEAT
  ;
```

Reference Section.

R1.0 General.

The terms and notation used in this section are based on those used by the FORTH-83 standard. They specify the effect on the stack, the parameters that each word takes on the stack and the context in which the word is intended to be used.

R1.1 General Notation.

For each word, the parameters passed to, and returned by, the word are described in the notation:

<before> – <after>

– where <before> are the stack parameters before execution.
<after> are the stack parameters after execution

In this notation, the top of the stack (ie. the most accessible item) is to the right of the parameter list, and applies at execution time. Where appropriate, the compile time stack notation is followed by: (compiling).

R1.2 Attributes.

Each word may be given one or more of the following attributes:

C	May only be used during the compilation of a colon definition
I	Indicates that the word is IMMEDIATE, and will execute during compilation, unless specific action is taken to avoid this.
V	User variable.
S	Required by the FORTH-83 Standard.
O	Part of an optional FORTH-83 Extension word set.
R	Part of the FORTH-83 Controlled Reference word set.

R1.3 Input Text.

Upper and lower case letters are distinct. The following notation is used:

<name>	An arbitrary FORTH word accepted from the input stream. Only the first 32 characters of a FORTH word are significant.
<ccc>	An arbitrary sequence of characters accepted from the input stream until the first occurance of the specified delimiting character. <ccc> can be from 0 to 255 characters in length.

The following are the stack parameter abbreviations and types of numbers used. They may be suffixed with a digit to differentiate multiple parameters of the same type. Unless otherwise noted, all stack parameters are 16-bit integers.

flag	0 for false, -1 for true.
8b	8 arbitrary bits (ie. a byte).
16b	16 arbitrary bits.
n	number -32768 to 32767.
+n	positive number 0 to 32767.
u	unsigned number 0 to 65535.
w	number -32768 to 65535.
addr	address.
32b	32 abitrary bits.
d	double number -2147483648 to 2147483647.
+d	positive double number 0 to 2147483647.
ud	unsigned double number 0 to 4294967295.
wd	double number -2147483648 to 4294967295.
sys	zero or more stack items used by the system.

Where the above are used in the Glossary of Words (section R2), their presence indicates how the numbers will be interpreted For example, in a multiply word, the presence of n as a result would indicate that a signed multiply was performed, whilst u would indicate that it was unsigned. n and u are otherwise identical, since they are both 16 bits and can both consist of any bit pattern.

The Enterprise specific words are listed later. Applications written using words taken ONLY from this list should be readily transferable to other IS-FORTH systems.
The following is a list of words available in IS-FORTH, in ASCII order.

!	**16b,addr –**	**S**

16b is stored at addr.

!CSP	**–**	

Stores the value of the stack pointer in the variable CSP. This is used during the compilation of a colon definition for checking for error checking.

"	**– addr** **– (compiling)**	**I**

Used in the form:

" <ccc>"

When used whilst interpreting, copies <ccc> to PAD and leaves addr pointing to the length byte of the string.

When used during compilation of a colon definition, compiles the string such that later execution will leave addr pointing to the length byte. Note that in this case the string at addr will be within the definition, and so should not be altered.

#	**+d1 – +d2**	**S**

The remainder of +d1 divided by BASE is converted to an ASCII character and appended to the output string (pointed to by the variable HLD). +d2 is the quotient maintained for further processing. Typically used between <# and #> to output a digit.

#>	**32b – addr,+n**	**S**

32b is dropped. addr is the address of the output string, and +n is its length (as required by TYPE). Typically used to end the conversion of a number to an ASCII string.

#S	**+d – 0,0**	**S**

is repeatedly performed until +d is 0. A single ASCII 0 is produced if +d is initially 0. Typically used between <# and #> to convert the whole number.

#TIB	**– addr**	**S,V**

A variable pointing to the number of bytes in the text input buffer.

$!	**addr1,addr2 –**	

The string at addr1 is copied to addr2. addr1 typically points to a string in PAD, and addr2 typically points to a string variable.

$. **addr –**

The string at addr is displayed. Equivalent to the sequence COUNT TYPE

$<> **addr1,addr2 – flag**

flag is false if the string at addr1 is the same as that at addr2. If flag is positive, then the string at addr1 is greater than that at addr2, otherwise the string at addr1 is less than that at addr2.

$@ **addr1 – addr2**

addr2 is the address in PAD of a copy of the string at addr1.

$CONSTANT **addr –**

A defining word used in the form:

" <ccc>" $CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, the address of a copy of the string at addr is left on the stack. If the same string is to be used more than once in an application, then $CONSTANT should be used, since there will then only be one copy of the string in memory, thus saving memory.

$VARIABLE +n –

A defining word used in the form:

+n $VARIABLE <name>

A dictionary entry for name is created, and +n bytes are allocated for the string variable. When <name> is subsequently executed, the address of the first byte of the string variable is left on the stack. The initial contents of the variable are not definied.

' **– addr** **S**

Used in the form:

' <name>

addr is the compilation address of <name>. An error is given if <name> is not found.

(**–** **S,I**
– (compiling)

Used in the form:

(<ccc>)

The characters <ccc>, delimited by), are comments and are ignored. Comments may be used anywhere a blank can occur during compiling or execution,provided that at least one space occurs
before the opening bracket and after the closing bracket.

(ABORT) -

Used in the form:

(ABORT) <name>

where name is a previously defined word. Causes <name> to be executed whenever an error occurs, except for errors caused by the ERROR word. If <name> returns, then ABORT will be executed. When <name> is executed, the current value in BLK is saved in SCR, the current value in >IN is saved in R#, and the error number is on the stack. To return to normal IS-FORTH error behaviour, the sequence (ABORT) ERROR can be used.

(BACKGROUND)

Used in the form:

(BACKGROUND) <name>

where <name> is a previously defined word. Defines <name> to be a word that executes as a background task to the main word being executed. The variable BACKGROUND contains a count which is decremented on each entry to the FORTH inner interpreter if FORTH is in the SLOW mode and BACKGROUND is already non-zero. When the count is decremented from one to zero, execution of the current word is suspended and <name> is executed. When <name> returns, execution of the current word continues.

It is the responsibility of <name> to reset the BACKGROUND variable count to a non-zero value immediately before it returns. The minimum value that BACKGROUND should be set to is 4, which results in maximum frequency of execution for <name>, and the maximum value for BACKGROUND is 65535, resulting in very infrequent execution.

Note that a background word should have no overall effect on the stack, nor should it alter the values of variables used by other words.

*****	**w1,w2 – w3**	**S**

w3 is the arithmetic product of w1 and w2.

***/**	**n1,n2,n3 – n4**	**S**

n1 is multiplied by n2, producing an intermediate 32-bit result which is then divided by n3, producing n4, the floor of the quotient.

***/MOD**	**n1,n2,n3 – n4,n5**	**S**

n1 is multiplied by n2, producing an intermediate 32-bit result which is then divided by n3, producing remainder n4 and the floor of the quotient n5.

+	**w1,w2 – w3**	**S**

w3 is the arithmetic sum of w1 and w2.

+!	**w1,addr –**	**S**

w1 is added to the w value at addr, and the result put back at addr.

+LOOP	**n –** **sys – (compiling)**	**S,C,I**

n is added to the loop index, and if this crosses the boundary between limit-1 and limit, then the loop is terminated and the control values dropped. Otherwise, execution continues at the word following the corresponding DO.

, **16b –** **S**

ALLOTs two bytes and stores 16b there.

w1,w2 – w3 **S**

w3 is the arithmetic difference between w1 and w2 (ie. w1-w2).

-! **w1,addr –**

w1 is subtracted from the value at addr, and the result put back at addr.

--> **–** **I,R**

When interpreting from a buffer, causes interpretation to continue with the next buffer. May be used within a colon definition that is split between two buffers.

-1 **– -1**

Leaves the constant -1. Definied as a word for speed and memory conservation.

-2 **– -2**

Leaves the constant -2. Defined as a word for speed and memory conservation.

-2ROT 32b1,32b2,32b3 – 32b3,32b1,32b2

The top three 32-bit stack entries are rotated, taking the top item to the deepest position. Rotates in the opposite direction to 2ROT and ROT.

-3 – -3

Leaves the constant -3. Defined as a word for speed and memory conservation.

-ROT 16b1,16b2,16b3 – 16b3,16b1,16b2

The top three stack items are rotated, taking the top item to the deepest position. Rotates in the opposite direction to ROT.

-TRAILING addr,+n1 – addr,+n2 S

+n1 is adjusted to produce +n2 such that trailing spaces are excluded from the string of +n1 characters starting at addr.

The value of n is displayed using the radix BASE in a free-field format.

– (compiling)

Used in the form:

." <ccc>"

Subsequent execution will cause the characters <ccdjusted to produce +n2 such that trailing spaces are excluded from the string of +n1 characters starting at addr.

The value of n is displayed using the radix BASE in a free-field format..

– from the next example.

Example.

Let us write a program to sort people into three categories, by age. Let us say that a child is someone under twelve, an adolescent is someone aged between twelve and eighteen and an adult is eighteen plus. Our sort program could look like this :-

```
: AGE DUP 12 < IF
                    ." Child"
                ELSE
                    18 < IF
```

– (compiling)

Used in the form:

." <ccc>"

Subsequent execution will cause the characters <ccc> (delimited by ") to be displayed. The space after ." and the final " are not part of <ccc> and are not displayed. May only be used in a definition. See .(.

– (compiling)

Used in the form:

.(<ccc>)

The characters <ccc> (delimited by) will be displayed. The space after

The value of n is displayed using the radix BASE right aligned in a field +n characters wide.

All the items on the stack are printed as signed 16-bit numbers, but are not removed from the stack. The left-most number is on the bottom of the stack, and the right-most on the top (ie. the most accessible).

/	**n1,n2 – n3**	**S**

n1 is divided by n2, producing the floor of the quotient n3.

/MOD	**n1,n2 – n3,n4**	**S**

n1 is divided by n2, producing the remainder n3 and floor of the quotient n4.

0	**– 0**	

Leaves the constant 0. Defined as a word for speed and memory conservation.

0!	**addr –**	

Stores 0 in addr.

0<	**n – flag**	**S**

The flag is true if n is negative.

0=	**w – flag**	**S**

The flag is TRUE if w is zero.

0>	**n – flag**	**S**

The flag is true if n is greater than zero.

1	**– 1**	

Leaves the constant 1. Defined as a word for speed and memory conservation.

1+	w1 – w2	S

w2 is the result of adding one to w1.

1+!	**addr –**	

Increments the 16-bit value at addr.

1-	**w1 – w2**	**S**

w2 is the result of subtracting one from w1.

1-!	**addr –**	

Decrements the 16-bit value at addr.

2	**– 2**	

Leaves the constant 2. Defined as a word for speed and memory conservation.

2!	**32b,addr –**	**O**

32b is stored at addr.

2* n1 – n2 **R**

n2 is the result of arithmetically shifting n2 left one bit.

2+ w1 – w2 **S**

w2 is the result of adding two to w1.

2- w1 – w2 **S**

w2 is the result of subtracting two from w1.

2/ n1 – n2 **S**

n2 is the result of arithmetically shifting n1 right one bit.

2@ addr – 32b **O**

32b is the value at addr.

2CONSTANT 32b – **O**

A defining word used in the form:

32b CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, 32b will be left on the stack.

2DROP **32b –** **O**

32b is removed from the stack.

2DUP **32b – 32b,32b** **O**

32b is duplicated.

2LITERAL **– 32b** **C,I**
32b – (compiling)

Typically used in the form:

[32b] 2LITERAL

Compiles a word that, when executed, leaves 32b on the stack. Unlike most other FORTHs, IS-FORTH will automatically compile a 32-bit number that appears within a colon definition.

2OVER 32b1,32b2 – 32b1,32b2,32b1 **O**

The second double number on the stack is copied to the top of the stack.

2ROT 32b1,32b2,32b3 – 32b2,32b3,32b1 **O**

The top three double number stack items are rotated, bringing the third item to the top of the stack.

2SWAP **32b1,32b2 – 32b2,32b1** **O**

The top two double numbers are exchanged.

2VARIABLE **–** **O**

A defining word used in the form:

2VARIABLE <name>

A dictionary entry for <name> is created, and four bytes allocated for the variable. When <name> is subsequently executed, the address of the variable is left on the stack. The initial value of the variable is not defined.

3 **– 3**

Leaves the constant 3. Defined as a word for speed and memory conservation.

: **–** **S**
– sys (compiling)

Used in the form:

: <name> ...;

which creates a word definition for <name> in the compilation vocabulary. The text from the input stream is then compiled (unless specific action is taken to prevent this). The newly created definition of <name> cannot be found in the dictionary until the terminating ; or ;CODE word. <name> is then called a "colon definition". An error is given if : is not balanced by ; or ;CODE.

;   –   C,I,S

sys – (compiling)

Stops compilation of a colon definition and sets the interpret state. Allows the <name> of the corresponding colon definition to be found in the dictionary.

;CODE   –   C,I,O

sys1 – sys2 (compiling)

Used in the form:

: <namex> ... <create> ... ;CODE ... END-CODE

where <create> is CREATE or any word that executes CREATE.

Stops compilation terminates the defining word <namex> and makes the CONTEXT vocabulary the assembler vocabulary. <namex> is subsequently used in the form:

<namex> <name>

which defines word <name>, which when executed will execute the machine code sequence between ;CODE and END-CODE. Thus ;CODE is similar to DOES>, execpt that the following words define a section of machine code. See section 5 "Assembler" and section 6 "Machine Code".

;S   –

When used in a block outside a colon definition, stops interpretation of the block.

< **n1,n2 – flag** **S**

flag is true if n1 is less than n2. Note that -32768 32768 < is true and that -32768 0 < is true.

<# **–** **S**

Initialises number conversion to ASCII. Typically used with #, #>, #S, HOLD and SIGN.

<> **w1,w2 – flag**

flag is true if w1 is not equal to w2.

<MARK **– addr** **C,O**

addr is an address related to the position in a colon definition of <MARK, and is typically later used by <RESOLVE to compile a backward branch address.

<RESOLVE **addr –** **C,O**

Compiles addr as the address of a branch after BRANCH or ?BRANCH. Typically used at the source of a backward branch, addr having been previously left by <MARK.

=	**w1,w2 – flag**	S

flag is true if w1 is equal to w2.

>	**n1,n2 – flag**	S

flag is true if n1 is grater than n2. Note that -32768 32768 > is false and that -32768 0 > is false.

>BODY	**addr1 – addr2**	S

addr2 is the parameter field address corresponding to the compilation address addr1.

>IN	**– addr**	**V,S**

addr is the address of a variable which contains the current character offset within the input stream.

>MARK	**– addr**	**C,O**

Used at the source of a foward branch. Typically used after ?BRANCH or BRANCH, compiling space in the dictionary for a branch address subsequently calculated by >RESOLVE using addr.

the variable CSP. Used for compilation error checking.

?DUP	**16b – 16b 16b** **or 0 – 0**	**S**

Duplicates 16b if not zero.

?STACK	**–**	

Gives an error if a stack overflow or underflow has occured. This is always checked anyway when executing words from the keyboard, and when exectuting compiled words when in the SLOW mode.

?TERMINAL	**– 8b**	

8b is the ASCII value of the key pressed on the keyboard, or 0 if no key has been pressed.

@	**addr – 16b**	**S**

16b is the value at addr.

ABORT	**flag –** **– (compiling)**	**S**

Clears the data stack and executes QUIT without displaying a message.

>R **16b –** **C**

16b is transfered to the return stack.

>RESOLVE **addr –** **C,O**

Used at the destination of a foward branch. Calculates the branch address (to the current dictionary location) using addr and places the branch address in to the space left by >MARK.

? **addr –**

Displays the 16-bit value at addr in a free-field format using the radix BASE.

?BRANCH **flag –** **C,O**

Used in the form:

COMPILE ?BRANCH

which compiles a conditional branch operation. If flag is false then execution continues at the address which immediately follows the ?BRANCH address in the dictionary. If flag is true then execution continues with the word immediately following the branch address. The branch address is typically generated by following ?BRANCH with <RESOLVE or >MARK.

?CSP **–**

Gives an error if the stack pointer value is not the same as the value in

ABORT"	**flag –** **– (compiling)**	**C,I,S**

Used in the form:

ABORT" <ccc>"

When subsequently executed, if flag is true, then the characters <ccc> are displayed, and ABORT is executed. If flag is false, then flag is dropped and execution continues with the word after <ccc>. The space after ABORT" and the final " are not part of <ccc>, and are not displayed.

ABS	**n – u**	**S**

u is the absolute value of n. Note that if n is -32768 then u has the same value.

ALLOT	**w –**	**S**

Allocates w bytes in the dictionary.

AND	**16b1,16b2 – 16b3**	**S**

16b3 is the bitwise AND of 16b1 and 16b2.

ASCII	**– 8b (interpreting)**	

Used in the form:

ASCII <ccc>

ASSEMBLER **–** **O**

Makes the assembler vocabulary the first vocabulary searched, ie. the CONTEXT vocabulary. For the extra FORTH words available in the ASSEMBLER vocabulary see section on "Assembler Vocabulary".

BACKGROUND **– addr** **V**

A variable containing the current count for the frequency of execution of the background word. See (BACKGROUND)

BASE **– addr** **S,V**

The value at addr is the current numeric conversion radix.

BEGIN **–** **C,I,S**
– (compiling)

Used in the form:

```
BEGIN ... flag WHILE ... REPEAT or
BEGIN ... flag UNTIL.
```

BEGIN marks the start of a loop.
A BEGIN ... UNTIL loop will execute until flag is true.
A BEGIN ... WHILE ... REPEAT loop will execute until flag is false.

BINARY **–**

Sets the value at BASE to 2.

BLANK addr,u – **R**

Sets u bytes starting at addr to the ASCII value for space. No action is taken if u is zero.

BLK – addr **S,V**

The value at addr is the number of the block from which the input stream is being taken. If zero, then the input stream is taken from the text input buffer.

BLOCK u – addr **S**

addr is the address of a buffer containing block number u. An error is given if u is not available. For details see section on "Cassettes, Disks and Buffers".

BRANCH – **C,O**

Used in the form:

COMPILE BRANCH

which compiles an unconditional branch operation to the address compiled immediately afterwards. The branch address is typically generated by <RESOLVE or >MARK.

BUFFER u – addr S

addr is the address of the buffer containing block u. The contents of the buffer at addr are not defined. For details see section on "Cassettes, Disks and Buffers".

C! 16b,addr – S

The least-significant 8 bits of 16b are stored at addr.

C, 16b – R

One byte is ALLOTted and the least-significant byte of 16b stored there.

C@ addr – 8b S

8b is the contents of the byte at addr.

CMOVE addr1,addr2,u – S

u bytes starting at addr1 are moved to addr2, starting with the byte at addr1. If u is zero, then no bytes are moved. CMOVE is used to move data down in memory to a lower address.

CMOVE> addr1,addr2,u – S

u bytes starting at addr1 u + 1- are moved to addr2 u + 1-, proceding to lower memory addresses. If u is zero then no bytes are moved. CMOVE>

is used to move data up in memory to a higher address.

CODE **– sys** **O**

A defining word used in the form:

CODE <name> ... END-CODE

Creates a dictionary entry for <name>, which is defined by the following sequence of assembler words. <name> cannot be found in the dictionary until END-CODE is executed. CODE makes the assembler vocabulary the CONTEXT vocabulary. See section on "Assembler Vocabulary" and section on "Machine Code".

COMPILE **–** **C,S**

Used in the form:

: <name> ... COMPILE <namex> ... ;

When <name> is executed, then the compilation address (code field address) of <namex> is compiled, instead of <namex> being executed.

CONCAT **addr1 – addr2**

Concatenates the string at addr1 with the string in PAD by copying the string at addr1 to the end of the string in PAD. addr2 is the start address of the resulting combined string in PAD, the length byte of which is adjusted to the length of the new string, limited to 255 if necessary.

CONSTANT **16b –** **S**

A defining word used in the form:

16b CONSTANT <name>

Creates a dictionary entry for <name> so that when <name> is later executed, 16b will be left on the stack.

CONTEXT **– addr** **O,V**

addr is the address of a user variable that specifies which vocabulary is to be searched first in the dictionary. Not normally altered by the user directly. See section on "Vocabularies".

CONVERT **+d1,addr1 – +d2,addr2** **S**

+d2 is the result of converting an ASCII number starting at addr1+1 into binary, using +d1 as the partial result. The value at BASE is used for this. addr2 is the address of the first unconvertable character.

COUNT **addr1 – addr2,+n** **S**

addr2 is addr1+1 and +n is the byte at addr1. Typically used to convert a pointer to a string with a byte count as the first byte into a pointer to a string with the byte count on the stack.

CR **–** **S**

Displays a carriage return and line feed.

CREATE **–** **S**

A defining word used in the form:

CREATE <name>

Creates a dictionary entry for <name>. The next available dictionary location is then the first byte of the parameter field of <name>, and when <name> is subsequently executed, this address is left on the stack.

CREATE-BUFFER **– sys**

Creates one or more new block buffers in memory. The number of new buffers created and the values of sys may vary from one implementation of IS-FORTH to another, and depends upon the machine on which it is run.

CSP **– addr** **V**

A variable which contains a previous value of the stack pointer. Used by !CSP and ?CSP for compiling errors.

CURRENT **– addr** **O,V**

The value of addr indicates to which vocabulary new dictionary entries are added. Not normally altered directly by the user. The value at addr is related to the vocabulary in the same way as the value at CONTEXT. See section on "Vocabularies".

D* **wd1,wd2 – wd3**

D* **wd1,wd2 – wd3**

wd3 is the 32-bit arithmetic product of wd1 and wd2.

D+ **wd1,wd2 – wd3** O

wd3 is the arithmetic sum of wd1 and wd2.

D- **wd1,wd2 – wd3** O

wd3 is the result of subtracting wd2 from wd1.

D. d – O

The double number d is displayed in a free-field format.

D.R d,+n – O

d is displayed using a radix of BASE right aligned in a field +n characters wide.

D0= **wd – flag** **O**

flag is true if wd is zero.

D2/ **d1 – d2** **O**

d2 is the result of d1 arithmetically shifted right one bit.

D< **d1,d2 – flag** **S**

flag is true if d1 is less than d2 according to the word < except extended to 32 bits.

D= **wd1,wd2 – flag** **O**

flag is true if wd1 equals wd2.

DABS **d – ud** **O**

ud is the absolute value of d.

DECIMAL **–** **S**

Sets the value at BASE to ten.

DEFINITIONS – S

The vocabulary in which new definitions are compiled is changed to that of the first vocabulary searched ie. the value at CURRENT is made the same as that at CONTEXT. See section on "Vocabularies".

DEPTH – +n S

+n is the number of 16 bit values on the stack before +n was placed on the stack.

DESTROY-BUFFERS sys –

Deletes from memory one or more block buffers, possibly freeing up the memory for other purposes. The number of buffers deleted and the values of sys may vary from one implementation of IS-FORTH to another, and depends upon the machine on which it is being run.

DMAX d1,d2 – d3 O

d3 is the greater of d1 and d2.

DMIN d1,d2 – d3 O

d3 is the lesser of d1 and d2.

DNEGATE d1 – d2 **S**

d2 is the two's complement of d1.

DO w1,w2– **C,I,S**
– sys (compiling)

Used in the form:

Do ... LOOP or
Do ... +LOOP

Begins a loop with an initial index value of w2 which terminates when the new index value computed by LOOP or +LOOP exceeds the limit value w1.

DOES> – addr **C,I,S**
– (compiling)

Defines the execution-time action of a word created by a high-level defining word. Used in the form:

: <namex> ... <create> ... DOES> ... ;

where <create> is CREATE or any word which executes CREATE.

Subsequently,when used in the form:

<namex><name>

the word <name> will be defined which, when executed, will place the address of it's parameter field on the stack and execute the sequence of words between DOES> and ; .

DP **– addr** **V**

A variable pointing to the next free byte in the dictionary.

DPL **– addr** **V,R**

A variable containing the number of characters converted during numeric conversion since the last fractional point or punctuation mark.

DROP **16b –** **S**

16b is removed from the stack.

DU. **du –**

du is displayed as an usigned number in a free-field format using the current radix in BASE.

DU< **ud1,ud2 – flag** **O**

flag is true if ud1 is less than ud2.

DUP **16b – 16b,16b** **S**

16b is duplicated.

ELSE — **C,I,S**

sys1 – sys2 (compiling)

Used in the form:

flag IF ... ELSE ... THEN

If the flag is false, then the words between ELSE and THEN are executed. If the flag is true, then the words after IF are executed until ELSE is executed, which then forces execution to continue after THEN.

EMIT **16b –** **S**

The least significant 8 bits af 16b are displayed.

EMPTY —

Empties all the user's dictionary, leaving the FORTH system with just the power-on words in the dictionary. Resets FENCE, (ABORT) and (BACKGROUND).

EMPTY-BUFFERS — **R**

Unassigns all block buffers in memory, marking them as empty. Does not save the contents to disk or casette.

END-CODE **sys –** **O**

Terminates a ;CODE or CODE definition and allows the corresponding

<name> to be found in the dictionary. Restores the CONTEXT vocabulary. See section 5 "Assembler Vocabulary" and section 6 "Machine Code".

ERASE **addr,u –** **R**

Sets u bytes starting at addr to zero. No action is taken ifu is zero.

ERROR **+n –**

Causes error number +n, but does not execute any error trapping word definied by (ABORT), so that any such word may use ERROR to cause the error in the normal way.

EXECUTE **addr –** **S**

The word whose compilation address (code field address) is addr is executed.

EXIT **–** **C,S**

When executed, forces execution to continue at the point indicated by the top item on the return stack. This is the word compiled by ; at the end of a colon definition, so that control is returned to the definition that originally passed control to the colon definition.

EXPECT **addr,+n –** **S**

Receives up to +n characters and stores from addr upwards. The actual number of characters received în put in SPAN.

FALSE **– 0**

Leaves the IS-FORTH representation of false on the stack, which is zero.

FAST **–**

Sets up IS-FORTH in its FAST mode. In this mode, stack overflow/underflow is not tested, the stop/break key is not tested for, no background words are executed and no interrupt words (if available) are executed. Time critical words can set up the FAST word at the start, and then back to the SLOW mode before returning.

FENCE **– addr** **V**

A variable containing an address within the user's dictionary below which FORGET will refuse to remove words. Initially set to just above the top power-on dictionary, but may be initialised to some other value using the form:

```
      ' <name> FENCE !
  or  HERE FENCE !
```

FILL **addr,u,8b –** **S**

u bytes of memory starting at addr are filled with 8b. No action is taken if u is zero.

FIND	**addr1 – addr2,n**	**S**

addr1 is the address of a string, length byte first, which is the name of a word to be searched for in the dictionary. If the word is not found, then n is zero (false) and addr2 is the same as addr1. If the word is found, the addr2 is the compilation address of the word and n is 1 if the word is immediate or -1 otherwise.

FLD	**– addr**	**V**

A variable containing the number of characters converted since the last execution of <# during numeric output conversion. <# sets FLD to zero.

FLUSH	**–**	**S**

Performs the same function as SAVE-BUFFERS, then marks all buffers as unassigned.

FORGET	**–**	**S**

Used in the form:

    FORGET <name>

Deletes <name> and all definitions added to the dictionary after <name>. An error is given if <name> does not exist or is below the address held in FENCE.

FORTH	**–**	**S**

Makes the FORTH vocabulary the CONTEXT ie. the first vocabulary searched in the dictionary. See section 4 "Vocabularies".

FORTH-83 – S

Does nothing to indicate that IS-FORTH conforms to the FORTH-83 Standard.

FORWARD – C,I

Used in the form:

```
: <name> ... FORWARD ... ;
```

Used in place of a word in the definition of <name> which has not yet been definied ie. allows <name> to include a forward reference to a word. The forward reference is subsequently resolved by using the word USES in the form:

```
<name> USES <namex>
```

where <namex> is the word (now definied) that FORWARD replaced in the definition of <name>. Allows mutual recursion between two or more words. Only one FORWARD is allowed in each definition. See the example infix expression evaluator in section 9 "Examples".

HERE – addr S

addr is the address of the next available dictionary location.

HEX	**–**	**R**

Puts the value 16 at BASE.

HOLD	**char –**	**S**

char is added to the end of the number conversion output string. Typically used between <# and #>.

HLD	**– addr**	**V**

A variable pointing to the next unused byte in the numeric output stream during numeric conversion. Successive digits produced by numeric output conversion are stored at decreasing memory addresses ie. a digit when converted is stored at HLD and the HLD is decremented.

I	**– w**	**S**

Obtains the current index of the inner-most DO...LOOP. Typically used in the form:

```
   DO ... I ... LOOP
or DO ... I ... +LOOP
```

IF	**flag –** **– sys (compiling)**	**C,I,S**

Used in the form:

```
   flag IF ... THEN
or flag IF ... ELSE ... THEN
```

If flag is true, then the words immediately following IF are executed. If an ELSE is executed, then execution continues after THEN. If flag is false, then execution continues after ELSE, or after THEN if no else is given.

May alternatively be used whilst interpreting, in which case IF cannot be nested, and the whole IF...THEN structure must be contained on one input line or one block. Typically used whilst interpreting in a block for conditional compilation or conditional assembly.

IMMEDIATE – **S**

Indicates that the most recently created dictionary entry is a word that is to be executed if encountered during compilation, instead of being executed.

INDEX +n1,+n2 –

Displays the top line of each block (which conventionally contains a comment describing the contents of the block) starting with block number +n1 and finishing with block number +n2.

INTERPRET – **R**

Starts text interpretation at the character indicated by >IN and BLK, until the input stream is exhausted.

J – w **S**

Obtains the current index of the inner-most-but-one DO...LOOP. Typically used in the form:

```
   DO ... J ... LOOP
or DO ... J ... +LOOP
```

KEY	**– 16b**	**S**

The least-significant 8 bits of 16b is the next character received. The character is not displayed.

LAST	**–**	**V**

A variable which points to the first byte of the last word CREATEd or defined.

LEAVE	**–** **–**	**C,I,S**

Used in the form:

```
   DO ... LEAVE ... LOOP
or DO ... LEAVE ... +LOOP
```

Transfers execution to the word following the LOOP or +LOOP. More than one LEAVE may appear in a loop.

LIST	+n –	**R**

Lists block number +n, without trailing spaces.

LITERAL	**– 16b** **16b – (compiling)**	**C,I,S**

Typically used in the form:
  [16b] LITERAL

Compiles a word which, when executed, will leave 16b on the stack.

LOAD	**+n –**	**S**

The contents of >IN and BLK (which define the input stream) are saved, and BLK is set to +n and >IN to 0. Interpretation then begins with block +n, and continues until either explicitly stopped or the input stream is exhausted. BLK and >IN are then restored.

LOOP	**–** **sys – (compiling)**	**C,I,S**

Equivalent to +LOOP with an increment value of one.

MAX	**n1,n2 – n3**	**S**

n3 is the greater of n1 and n2 according to the word > .

MIN	**n1,n2 – n3**	**S**

n3 is the lesser of n1 and n2 according to the word < .

MOD n1,n2 – n3 S

n3 is the remainder of dividing n1 by n2.

NAND 16b1,16b2 – 16b3

16b3 is the bitwise NAND between 16b1 and 16b2.

NEGATE n1 – n2 S

n2 is the two's complement of n1.

NOOP –

Does nothing.

NOR 16b1,16b2 – 16b3

16b3 is the bitwise NOR between 16b1 and 16b2.

NOT 16b1 – 16b2 S

16b2 is the one's complement of 16b1.

NUMBER addr – d

Converts the string (length byte first) at addr into number d, using the current BASE value. If no conversion is possible, then an error is given.

OCTAL – R

Sets the value at BASE to 8.

OR 16b1,16b2 – 16b3 S

16b3 is the bitwise OR of 16b1 and 16b2.

OUT – V

A variable which is incremented by EMIT and characters output using words that use EMIT. OUT is incremented for every ASCII character in the range 20H (space) to 126H (¯), and set to zero for a carriage return (0DH). Other characters do not affect OUT.

OVER 16b1,16b2 – 16b1,16b2,16b1 S

The second item on the stack is copied to the top of the stack.

P! **8b,+n –**

8b1 is stored at Z80 port +n.

P@ **+n – 8b**

8b is the byte read from Z80 port +n.

PAD **– addr** **S**

addr is the start address of an area of scratch pad RAM which is over 256 bytes long. The contents of this area of memory may be lost if anything is added to the dictionary.

PICK **+n – 16b** **S**

16b is a copy of the +nth item on the stack, not including +n itself. Thus

0 PICK is equivalent to DUP
1 PICK is equivalent to OVER

Note that PICK when used as above is less efficient than the equivalent words.

PREV **– addr**

addr is the address of the most recently accessed block buffer.

QUIT — **S**

Clears the return stack, sets the interpret state, excepts input text and begins interpretation.

R# – addr

A variable which may be used to contain the current cursor position for words which edit blocks. Currently used to save >IN after an error if an (ABORT) word is defined or if the error occured whilst loading (BLK non-zero).

R> – 16b C,S

16b is a value which has been removed from the top of the return stack and placed on the data stack.

R@ – 16b C,S

16b is a copy of the top item on the return stack.

RECURSE – C,I,R

Compiles the compilation address of the word being defined.

REPEAT — C,I,S
sys – (compiling)

Used in the form:

BEGIN ... flag WHILE ... REPEAT

At execution time, execution continues at the point immediately following BEGIN.

RND +n1 – +n2

+n2 is a pseudo-random number from 0 to +n1-1 inclusive. If +n1 is zero then +n2 is in the range 0 to 65536 inclusive.

ROLL +n – S

The +nth item on the stack is removed (the rest of the stack items being moved to fill the vacated position) and then placed on top of the stack. No action is taken if +n is zero. Thus:

2 ROLL is equivalent to ROT

Note that ROLL is less efficient than the equivalent operation.

ROT 16b1,16b2,16b3 – 16b2,16b3,16b1 S

The top three stack items are rotated, bringing the third item to the top of the stack.

RP! **–**

Initialises the return stack. Any words using this must not return !.

S->D **n – d**

Sign-extends 16-bit number n to convert it into a 32-bit number d.

SAVE-BUFFERS **–**

The contents of all UPDATEed block buffers are written to cassette or disk. For further details see section 8 "Cassettes, Disks and Buffers".

SCR **– addr** **V**

A variable which may contain the current block number for block editing commands. Set by LIST to the block number being listed. BLK is saved in SCR after an error if an (ABORT) word has been defined or if the error occured during loading (BLK non-zero).

SIGN **n –** **S**

If n is negative, then a minus sign is appended to the current output string. Typically used between <# and #>.

SLICE **n1,n2 – addr**

addr is an address within the string currently in PAD (which will be destroyed by the SLICE operation) and points to a string containing the characters from the n1th character to the n2th character of the original string inclusive. If n2<n1 then addr points to an empty string. If n1<1 then the first character of the original string is assumed. If n2>length of string then the last character is assumed.

SLOW –

Sets up IS-FORTH in its SLOW mode. In this mode, stack overflow/underflow is tested for, the stop/break key is tested for and background and interrupt words (if available) will be executed. For maximum speed the FAST mode should be used.

SMUDGE –

Toggles a bit in the last word CREATEd or defined so that it cannot be found in dictionary searches. Another SMUDGE will toggle the bit again.

SP! –

Clears the stack.

SPACE – S

Outputs a space.

SPACES	**+n –**	**S**

Outputs +n spaces. No action is taken if +n is zero.

SPAN	**– addr**	**V,S**

The value at addr is a count of the number of characters actually received and stored by the last execution of EXPECT.

STATE	**– addr**	**V,S**

If the value at addr is non-zero, then compilation is occuring, otherwise exection is occuring.

SWAP	**16b1,16b2 – 16b2,16b1**	**S**

The top two stack entries are exchanged.

THEN	**–**	**C,I,S**
	sys – (compiling)	

Used in the form:

```
    flag IF ... THEN
or  flag IF .. ELSE ... THEN
```

Execution continues after THEN after ELSE or after IF when no ELSE is present.

THRU +n1,+n2 –

The block numbers from +n1 to +n2 inclusive are loaded as with LOAD.

TIB – addr V,S

addr is the address of the text input buffer.

TRUE – -1

Leave the value that IS-FORTH uses to represents true, which is -1.

TYPE addr,+n – S

+n characters are displayed starting with the character at addr. No action is taken if +n is zero.

U. u – S

u is displayed as an unsigned number in a free-field format.

U< u1,u2 – flag S

flag is true if u1 is less than u2.

UM*	**u1,u2 – ud**	**S**

ud is the 32-bit product of n1 and n2. All values are unsigned.

UM/MOD	**ud,u1 – u2,u3**	**S**

u2 is the remainder and u3 the quotient after dividing the 32-bit ud by u1. All values are unsigned.

UNTIL	**flag –** **sys – (compiling)**	**C,I,S**

Used in the form:

```
BEGIN ... flag UNTIL
```

If flag is false, then execution continues with the word following BEGIN, otherwise the loop is terminated and execution continues with the word following UNTIL.

UPDATE	**–**	**S**

Marks the block buffer most recently accessed as being updated.

USES	**sys –**	

Used in the form:

```
<name> USES <namex>
```

to resolve forward references created by FORWARD. See FORWARD.

VARIABLE – S

A defining word used in the form:

VARIABLE <name>

A dictionary entry for <name> is created, and two bytes allocated for the variable. When <name> is subsequently executed, the address of the variable is left on the stack. The initial value of the variable is not defined.

VLIST –

Lists all the words in the current search order. On power-up, this will be all the pre-defined words in the FORTH vocabulary. After executing ASSEMBLER this will be all the words in the ASSEMBLER vocabulary followed by all the words in the FORTH vocabulary. Words are displayed with two spaces separating them except where there is a change in vocabulary, in which case three spaces are displayed.

VOCABULARY – S

A defining word used in the form:

VOCABULARY <name>

A dictionary entry for <name> is created which specifies a new dictionary search order. When <name> is subsequently executed, the vocabulary <name> becomes the first searched in the dictionary (ie. becomes the CONTEXT vocabulary). If <name> becomes the compilation vocabulary (ie. becomes the CURRENT vocabulary). then subsequent new dictionary definitions are appended to the vocabulary <name>. See section on "Vocabularies".

VOC-LINK — **addr**

A variable which points to a field within the most recently created vocabulary. This vocabulary in turn contains a field which points to the previously defined vocabulary. Thus all vocabularies are linked in chronological order so that FORGET can FORGET truough multiple vocabularies. VOC-LINK is unlikely to be used by the user.

WHILE **flag –** **C,I,S**
sys1 – sys2 (compiling)

Used in the form:

BEGIN ... flag WHILE ... REPEAT

When flag is false, execution continues with the word following REPEAT. When flag is true, execution continues with the words between WHILE and REPEAT, which then continues execution at the word following BEGIN.

WORD **char – addr** **S**

Returns a string, length byte first, at addr. The characters for the string are obtained starting at the current position in the input stream, and ending with the delimiting character char, which is not included in the length, or with the end of the input stream. Leading delimiter characters are ignored. If the input stream is already empty when WORD is called, then a null string will be returned (ie. a string with a length of 0).

>IN is adjusted appropriately.

The string may become invalid if anything is added to the dictionary. Note that the text interpreter may also use this area.

XOR **16b1,16b2 – 16b3** **S**

16b3 is the bitwise exclusive-OR of 16b1 and 16b2.

[**–** **I,S**
– (compiling)

Sets the interpret state. Typically used to cause the execution of words during compilation.

['] **– addr** **C,I,S**
– (compiling)

Used in the form:

['] <name>

Compiles the compilation address of <name> as a literal, so that when the definition in which ['] is being used is subsequently executed, addr is left on the stack.

[COMPILE] **–** **C,I,S**
– (compiling)

Used in the form:

[COMPILE] <name>

Forces the compilation of word <name>. This ensures that <name> will always be compiled, even if it is an immediate word that would otherwise always have been executed.

] — S

Sets the compilation state. Typically used to resume compilation after [has been used.

Enterprise-Specific Forth Words

R4.1 Introductory Notes.

Enterprise FORTH is based on IS-FORTH, with many additional words added to take advantage of EXOS and the Enterprise hardware.

Through the use of block buffers in RAM, Enterprise FORTH is able to take advantage of extra memory available outside that addressable by the Z80.

The terms and notation used in this document are similar to those used in the IS-FORTH Specification, with the following extra stack parameters defined:

e	EXOS variable number. 0 to 255.
c	Channel number 0 to 255. c=255 refers to the 'default channel' set up in the EXOS variable.

Enterprise Forth uses a small redirection table for the extra words that use channels (see section on "Channels"). The terms 'current text channel', 'current graphics 'current graphics channel', 'current keyboard channel' and 'current editor channel' all refer to the channels contained in this redirection table.

#EDITOR **c –**

c is the channel to be used for subsequent operations to the editor. See section on "Channels".

#GRAPHICS **c –**

c is the channel to be used for subsequent graphics operations. See section on "Channels".

#STATUS **c – n**

Reads the status of channel c. n is 0 if a character is ready to be read, 255 if end of file, otherwise 1.

#TEXT **c –**

c is the channel to be used for subsequent text operations. See section on "Channels".

(SWI) **–**

Used in the form:

```
(SWI) <name>
```

where <name> is a previously defined word. The currently executing word is interrupted and execution continues with <name> when a Software Interrupt occurs from EXOS. When <name> returns, execution of the originally executing word resumes. The software interrupt may be from any source (eg. a key press, the network) except the STOP key, which always causes execution of the current word to halt.

******* –

Defined to do nothing so that pressing the 'enter' key when the cursor is on an error message does not produce another error message.

+DRAW **n1,n2 –**

Turns the graphics beam on, and moves it to (x+n1,y+n2) where (x,y) is the current beam co-ordinate, thus drawing a line relative to the current beam position. n1 or n2 may be negative. Uses the current graphics channel.

+MOVE **n1,n2 –**

Turns the graphics beam off, and moves it to (x+n1,y+n2) where (x,y) is the current beam co-ordinate, thus moving the beam relative to its current position. n1 or n2 may be negative. Uses the current graphics channel.

+PLOT **n1,n2 –**

Turns the graphics beam off, moves it to (x+n1,y+n2) and turns it on again where (x,y) is the current beam co-ordinate, thus plotting a point relative to the current beam position. n1 or n2 may be negative. Uses the current graphics channel.

-TEXT **+n –**

Sets up and displays 24 lines of text with +n columns (40 or 80). Resets the current graphics and text channel numbers. Closes the graphics channel if open.

ASK **e –8b**

8b is the value in EXOS variable e. This word is equivalent to E@

AT **+n1,+n2 –**

Positions the editor cursor at row +n1, column +n2. +n1 = +n2 = 1 positions the cursor at the top left of the editor buffer / video page. If +n1 or +n2 are 0 then that co-ordinate remains the same. Uses the current editor channel.

ATTRIBUTE **–**

Sets up and displays 20 lines of attribute graphics, with 4 lines of text below. Resets the current graphics and text channel numbers.

ATTRIBUTES **8b –**
Sets the attribute flags to 8b. Uses the current graphics channel.

BLACK **– 0**

The absolute colour number for black.

BLUE **– 36**

The absolute colour number for blue.

CAPTURE **c1,c2 –**

Causes subsequent reads from channel c1 (which must exist) to come instead from channel c2.

CHARACTER n1,n2,n3,n4,n5,n6,n7,n8,n9,n10 –

Re-defines character n10. n9 is the bottom row of the character. Uses the current text channel.

CLOSE **c –**

Closes channel c.

COLOR **+n1,+n2 –**

Changes palette colour +n2 to colour +n1. As COLOUR. Uses the current graphics channel.

COLOUR **+n1,+n2 –**

Changes palette colour +n2 to colour +n1. Uses the current graphics channel.

CREATE-BUFFERS **–**

Attempts to allocate a 16K segment of RAM from EXOS, and uses it for 16 extra block buffers. An error is given if an extra whole segment is not available. Upto 16 segments may be allocated in this way. Since there are eight buffers always present in the system, a total of 264 buffers are thus possible (but unlikely !).

CYAN **– 182**

The absolute colour number for cyan.

DATE! **addr –**

Sets the date counter using the string at addr. The string must contain 8 characters and be in the form "YYYYMMDD" ie. according to ANSI Standard X3.30.

DATE@ **– addr**

Returns the current date counter value as a string in PAD at addr. The string contains 8 characters and is in the form "YYYYMMDD" ie. according to ANSi Standard X3.30.

DESTROY-BUFFERS **–**

All buffers previously created by CREATE-BUFFERS are destroyed, and their RAM released back to EXOS. An error is given if any of the buffers are updated, thus reducing the risk of accidentally loosing all source code.

DEVICE addr –

Sets the default device or unit number to the string at addr (length byte first). If the string contains just a number, then only the default unit number is affected. If the string contains a device name but no unit number, then the default unit number is set to 0. Initially, the default device is TAPE on a cassette system or DISK on a disk system. The default device and unit numbers are used in any channel open operation with a filename that does not specify a device and when BUFFERS OFF is in effect and FORTH writes out a block.

DISPLAY +n1,+n2,+n3,c –

Displays a video page opened as channel c. +n1 is the first line in the page to display, +n2 is the first line on the screen, and +n3 is the number of lines to display. If +n1 is 0, then border colour is displayed over the specified range.

DRAW +n1,+n2 –

Turns the graphics beam on, and moves it to (+n1,+n2), thus drawing a line. Uses the current graphics channel.

DURATION +n –

Sets the duration for the next SOUND performed in 1/50ths of a second. See SOUND and the EXOS documentation for more details.

E! **8b,e –**

8b is stored in EXOS variable e.

E@ **e – 8b**

8b is the value in EXOS variable e.

EDIT **+n –**

Begins the editing of block number +n. See section on "IO, Cassettes, Disks and Buffers".

EFFECTS **8b –**

Sets the effects byte for the next SOUND performed. See SOUND and the EXOS documentation for more details.

ELLIPSE **+n1,+n2 –**

Draws an ellipse with an x radius of +n1 and a y radius of +n2, with the centre at the current beam position. Uses the current graphics channel.

ENVELOPE **+n –**

Defines envelope number +n (0 to 254). The envelope consists of upto forty separate phases specified previously by PHASE and RELEASE. After performing ENVELOPE, the next PHASE defines the first phase of the envelope defined by the next ENVELOPE. See the EXOS documentaion for more details.

EXT addr –

The string at addr (length byte first) is given to EXOS to pass around the ROMs in the system, and usually starts up a new applications program (thus loosing anything put into memory during the FORTH session) or performs some 'service'. See the EXOS specification.

FKEY $,+n –

Programs function key +n with $. If $ = "" then the function key will cause a software interrupt when pressed. Uses the current keyboard channel.

FONT –

Resets the character font to the default characters. The pound sign is re-programmed to be a hash since many FORTH words use a hash. Uses the current text channel.

FREE – +n

+n is the number of bytes available for use by the dictionary or stack.

GET c – 8b

8b is the byte read from channel c.

GRAPHICS –

Displays the default graphics page, if open. If it is not open or has been re-opened by the user with the wrong size, then the default graphics page is closed and re-opened with the same colour mode and resolution as the last page opened, or 4 HIRES initially.

GREEN – 146

The absolute colour number for green.

HIRES +n –

Sets up and displays 20 lines of high resolution graphics, with 4 lines of text below. +n is the colour mode (2, 4, 16 or 256). Resets the current graphics and text channel numbers.

INFO –

Displays information about the current memory usage. The total number of bytes of RAM (working + non-working) is displayed followed by the number not working (if non-zero) and the number of bytes available for use for buffers or the dictionary and stack. Following this the block number and the first 24 characters of the first line of all blocks in memory are displayed with a '*' next to the block number if updated. If any blocks have been updated, then the number of buffers in the system and the number of buffers free (ie. not updated) is also displayed.

INK +n –

Sets the current plotting colour to palette colour +n. Uses the current graphics channel.

INTERRUPT –

Causes the next SOUND performed to interrupt any previous sounds in the queue. See SOUND and the EXOS documentation for more details.

JOY +n – 8b

Reads the internal or external joystick, according to +n (0 => internal, 1=> external 1, 2=> external 2). Bits 0 to 4 of 8b are set according to the direction and 'fire' button (or space bar for the internal joystick). Bits 0,1,2,3,4 are set for right, left, up, down and fire respectively. Uses the current keyboard channel.

LEFT +n –

Defines the volume of the left sound channel for the next SOUND performed. See SOUND and the EXOS documentation for more details.

LORES +n –

Sets up and displays 20 lines of low resolution graphics, with 4 lines of text below. +n is the colour mode (2, 4, 16 or 256). Resets the current graphics and text channel numbers.

MAGENTA – 109

The absolute colour number for magenta.

MOVE +n1,+n2 –

Turns the graphics beam off, and moves it to (+n1,+n2). Uses the current graphics channel.

NAME addr –

Defines the filename used for writing out blocks with SAVE-BUFFERS when BUFFERS ON is in effect. If BUFFERS OFF is in effect, then the blocks are writen out as separate files with the filename derived from the block number. The latter is usually used with disks and the former with cassettes.

ON e –

Sets the EXOS variable e to 0, which usually turns an EXOS feature on.

OFF e –

Sets the EXOS variable e to 255, which usually turns an EXOS feature off.

OPENIN addr,c –

Opens channel c for input using the string at addr as the file name (length byte first).

OPENOUT addr,c –

Opens channel c for output using the string at addr as a filename (length byte first).

PAINT –

Fills an area of the graphics screen from the current beam position in the current ink colour up to any boundary which is not the same colour as the colour at the current beam position. Uses the current graphics channel.

PALETTE n1,n2,n3,n4,n5,n6,n7,n8 –

Defines a new palette of colours. n8 is palette colour 0. Uses the current graphics channel.

PAPER +n –

Sets the current background colour to +n. This will take effect when the video page is next cleared. Uses the current graphics channel.

PHASE n1,n2,n3,n4 –

The sound envelope phase described by n1,n2,n3 and n4 is added to the list of phases used to define an envelope with the next ENVELOPE performed. If a RELEASE has been performed since the last ENVELOPE, then the phase defined belongs to the release part of the envelope. See ENVELOPE and the EXOS documentation for more details.

PITCH +n –

Defines the pitch for the next SOUND performed. See SOUND and the EXOS documentation for more details.

PLOT +n1,+n2 –

Turns the graphics beam off, moves it to (+n1,+n2), and turns the beam on again, thus plotting a point. Uses the current graphics channel.

PLOTTING +n –

Defines the plotting mode for the current graphics channel. +n is 0, 1, 2 or 3 for PUT, OR, AND and XOR plotting respectively.

PUT 8b,c –

Writes byte 8b to channel c.

REDIRECT c1,c2 –

Causes subsequent writes to channel c1 (which must exist) to go instead to channel c2.

RGB n1,n2,n3 – n4

n1,n2 and n3 are amounts of red, green and blue respectively (0 to 7), and n4 is an 'absolute' colour (such as that used by PALETTE and BORDER) in the range 0 to 255.

RIGHT +n –

Defines the volume of the right sound channel for the next SOUND performed. See SOUND and the EXOS documentation for more details.

RND +n1 – 16b

Returns a pseudo-random number between 0 and (+n – 1) inclusive. If +n is zero, then 16b is between 0 and 65535.

S! 8b,addr,s –

8b is written to addr in segment s. addr is ANDed down into an address in the first 16K (ie. an offset into segment s). Note that S! corresponds to C! rather than !, in that only one byte is written.

S@ addr,s – 8b

8b is the value at addr within segment s/ addr is ANDed down into an address in the first 16K (ie. an offset into segment s). Note that S@ corresponds to C@ rather than @, in that only one byte is read.

SET 8b,e –

8b is stored in EXOS variable e. This word is equivalent to E!

SOUND +n –

Starts a sound using envelope +n. The sound is defined by the previously specified PITCH, DURATION, LEFT, RIGHT, EFFECTS, SOURCE, SYNC and INTERRUPT. After performing a SOUND, any subsequent PITCHes etc. refer to the following SOUND. See the EXOS documentation for more details.

SOURCE +n –

+n is the sound source used for the next SOUND performed. See SOUND and the EXOS documentation for more details.

SPFUNC 8b1,16b1,8b2,c – 8b3,16b2,8b4

Performs an EXOS special function call number 8b2 to channel c. 8b1 is passed in register C and 16b1 in register DE. 8b3 and 16b2 are the results in registers C and DE respectively, and 8b4 is the status returned by EXOS (ie. zero for no error, else error number).

STYLE +n –

Sets the current plotting style to +n. 0 and 1 are continuous lines and other values are various types of broken lines. Uses the current graphics channel.

SYNC +n –

Defines the synchronisation byte for the next SOUND performed. See SOUND and the EXOS documentation for more details.

SYSLOAD addr –

addr is the address of a string (length byte first) which is used as a filename to load a system extension via EXOS.

TEXT –

Attempts to display 24 lines of text. If this is not possible (because ATTRIBUTE, HIRES etc. have opened 4 lines of text) then the graphics channel is closed and 24 lines of text are opened and displayed.

TIME! addr –

Sets the time counter from the string at addr. The string must contain 8 characters and be in the form "HH:MM:SS" ie. according to the ANSI Standard X3.43.

TIME@ – addr

Reads the current value of the time counter as a string in PAD at addr. The string contains 8 characters in the form "HH:MM:SS" ie. according to ANSI Standard X3.43.

TOGGLE e –

Complements the contents of EXOS variable e, which usually toggles the state of an EXOS feature.

VADDR – addr1,addr2

Returns addresses related to a video page. For further details see the EXOS/video driver specification. addr1 and addr2 correspond to Z80 registers BC and DE respectively. Uses the current graphics channel.

WHITE **– 255**

The absolute colour number for white.

WIPE **+n –**

Fills block number +n with spaces.

YELLOW **– 219**

The absolute colour number for yellow.

ok **–**

Defined as a word (which sends the editor the necessary characters to delete the 'ok') so that if the 'enter' ket is pressed when the cursor is over an 'ok' previously printed by FORTH, then an error will not occur.