ORIC-1

FORTH PROGRAMMING MANUAL

WARNING

ORIC–FORTH

Contents

INTRODUCTION

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, Virginia, USA. It was created out of dissatisfaction with available programming tools, especially for observatory automation.

Mr. Moore and several associates formed FORTH Inc., in 1973 for the purpose of licensing and support of the FORTH operating system and Programming Language, and to supply application programming to meet customers requirements.

This version of FORTH is that issued by the Forth Interest Group (FIG) which is centred in northern California. The group was formed in 1978 by FORTH programmers to encourage use of the language, and interchange of ideas through seminars and publications.

FIG issue a language model, and code implementations for a number of processor types, the publications being in the public domain; however, each requires customisation to a particular target system. Customised installations are the property of the customiser, who then holds copyright for his/her particular version.

This handbook does not set out to be an exhaustive text-book on the language, merely an introduction to its use, and a general description of the internal workings. Users seeking further information of FORTH may like to consider joining the Forth Interest Group, whose address is: P.O. Box 1105, San Carlos, California 94070, U.S.A.

FIG hold a number of books, some of which may be purchased in this country through bookshops, and they should be happy to send you a membership form/publications list.

A good book now available is 'Starting FORTH', by Leo Brodie, published by Prentice-Hall; though this is based on polyFORTH*, which has some differences to figFORTH.

*polyFORTH is a trademark of FORTH Inc.

The Source Tape

Thank you for purchasing this Forth package for your Oric 1 computer system. We hope you will get both pleasure and an improved knowledge of programming techniques out of this unusual language.

The first thing to be done is to load the Forth program into your Oric. Note that all this is done simply by entering: CLOAD "FORTH",S, the files on the distribution tape have been recorded at the slow speed to ensure that you can read them reliably. If you wish you could make your own copy at the fast speed, how to do this will be explained later. Forth will take about 10 minutes to be read in.

Once the program has been loaded you can enter Forth by typing CALL #400 (Return). If you accidentally fall back into Basic (for example, by pressing the Reset button) you can re-start Forth without losing all your work with: CALL #404 (Return)

On entering Forth you will be greeted with the message: ORIC–FORTH V1 OK. You should then enter the command: EMPTY– BUFFERS (Return). This command initialises the cassette buffers (explained later on). Failing to do this may result in I/O being blocked.

If you now type VLIST, Forth will list out every command (or word) that it can understand. Control-C will stop it for you.

Also try the following: 2 3 * . (Return) Putting a space between each item. The answer 6 should be printed to the right of the dot. What you have done is to give the Forth interpreter two numbers, 2 and 3. These are pushed onto the stack as they were entered. The * (multiply) operator then multiplies these two stack entries replacing them with the answer on the top of the stack. Finally the command '.' (Dot) takes the number on the top of the stack and prints it on the screen.

If you now type { . CR } i.e. the instructions inside the curly brackets you will get the error message "EMPTY STACK" since there should be nothing there to print.

Cassette I/O

FORTH is designed to work with disc memory for bulk storage, where the disc is handled to provide virtual memory. This means that the disc looks as though it is an extension of normal memory. The FORTH method is to imagine the disc as consisting of 1K byte blocks numbered from 0 to N, the capacity of a disc. If the user requests that BLOCK 3 is to be 'used', then this block of 1K is fetched from the disc into a buffer in RAM. If the RAM buffer already contains another block from disc, then the RAM buffer is written back to disc before the new block is fetched. (Actually it is only written back if it has been modified).

In this simple manner, the whole of the disc is accessible in a direct manner, quick and simple to use. If the user requests a block it appears in the buffer. The fetch/rewrite operations are automatic, and are carried out by a group of FORTH Words collectively known as the Forth Virtual Disc Manager.

These 1K byte blocks are also known as SCREENS because they can be displayed (on normal VDU) as one screen full, 16 rows of 64 characters. This is not quite true for the Oric!

Oric Cassette Adaption

To allow a similar system with cassette which neither sacrifices the speed and flexibility of this mechanism, nor makes this version of FORTH incompatible with a future update to disc, the following method has been used.

A 7K byte block of RAM has been reserved as a 'micro-disc. The normal disc manager routines pretend this 7K block is a disc and they fetch/update 128 byte sectors of the normal disc buffer.

The cassette commands now load, and save the 7K microdisc in units of 1K SCREENS, which can be manipulated by Forth in the usual way, with no restriction on whether they contain source text, data, or whatever the user wishes.

Cassette Commands

CLOAD loads 1K Screens from tape to the 'micro-disc' buffers. The start and end screen number must be on the stack, for example:

1 3 CLOAD CR loads Screens 1, 2 and 3 from tape.
1 3 CSAVE similarly dumps them.
SPEED is a variable to control the tape speed.
Storing 0 here sets FAST
Storing 1 here sets SLOW
For example:- 0 SPEED ! (CR) sets fast

Note that CLOAD, CSAVE, and SPEED are like all FORTH commands and can be invoked either from the keyboard, or from within a program.

The full contents of the cassette are listed in the appendix.

Chapter 2

The Four(th) Fundamentals

These are as follows:

* The two stacks
* Post-fix notation
* The Dictionary
* Virtual Memory

Before getting to grips with the language it is essential to have a grasp of each of the above ideas. If you have used a Hewlett-Packard calculator, the first two items will be familiar.

The Stacks

FORTH maintains two push down stacks of numbers – last in, first out type. These may be pictured as a vertical spring loaded rack. As you push a new item into the rack, you push down all the existing items that are in it, putting the new one on the top. Taking the top item off then lets all the others 'POP UP' to reveal the next item available.

This is illustrated in Fig 1. It is very important to notice that the most RECENT item added is the FIRST one out again.

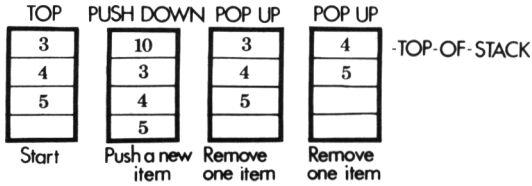


Fig. 1. A push-down stack

Because of the way most microprocessors and computers work, inside the machine memory, the stacks usually work 'upside down' to this description; that is new items are added/removed from the low end of the stack, as shown in Fig 2.

Hi Memory Address

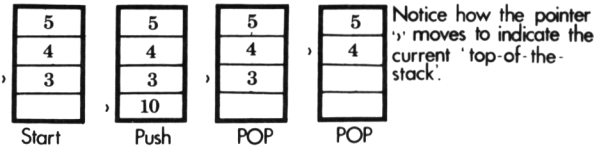


Fig. 2.

Which way up you wish to visualise the stacks does not really matter; most people visualise them as in Fig 1.

The Return Stack

This is the conventional subroutine return stack, and for the 6502 it occupies the address range Ø1FF down to Ø1ØØ (hex). FORTH stores linkage information here in the usual way, and also some other items from time to time. The processor stack pointer S does the work of the pointer '>' shown in Fig 2. This stack will always be referred to by its full name 'the return stack'.

The Parameter Stack

All calculations and operations, are carried out upon items in place at or near the top of this stack. The items are nearly always 16 bit integers, though double precision (32 bit) numbers may also be used. Manipulation of single bytes takes place as 16 bit integers with the unused high bits held at zero.

For example, the operator + adds two integers together. It removes the top two numbers from the stack, adds them, and returns ONE 16 bit integer to the stack as the answer.

It is possible to transfer numbers from one stack to the other, (with care), and to manipulate the relative positions of the top 3 or 4 items.

The parameter stack occupies most of the zero page and uses the processor X register as the stack pointer.

Postfix Notation

This is the concept of supplying an arithmetic operator AFTER the parameters. It is most easily seen with a simple arithmetic example.

Consider the following two statements:-

3 + 2 =	Algebraic notation
3 2 +	Postfix

The first is the 'normal' way, the second one reads as follows:

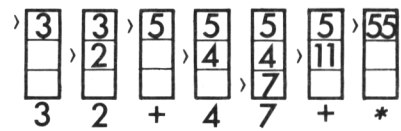
Reading from left to right, FORTH first encounters the 3. This, like all explicit numbers, is pushed onto the stack. Likewise the next item, the 2. The next thing is + which FORTH interprets as follows:

'Take the top two numbers from the stack, add them, and return the answer to the stack'.

This is much quicker than the algebraic form, where the line must be scanned beyond the operator (+ in this example) to ensure that all the variables or arguments have been located.

The Postfix notation requires that the arguments exist (on the stack) before the operation is invoked.

Try this example: 3 2 + 4 7 + * which yields 55. See the illustration of the stacks below.



Notice how the intermediate products 5 and 11 are left on the stack and appear ready for the multiply operator without any intervention.

In normal (algebraic) fashion, this would have been written
(3+2) * (4+7) =

Note also how the postfix version did not require brackets. This is because the order in which the operations take place is fixed only by the order in which YOU present them, NOT by some arbitrary rules of priority.

I hope you can now see how the postfix idea falls in perfectly with a stack operating machine, and how it increases the efficiencey and throughput of the program. It may surprise you that many languages are actually postfix 'inside' and they spend much code and time in converting from algebraic notation and back again for the supposed 'convenience' of the user. FORTRAN compilers fall into this category.

The use of a stack also means less named variables for storing intermediate results, or for passing arguments between routines. Arguments for routines are passed on the stack, of course. Named variables and constants can be created if required, but in general, the fewer the better.

The Dictionary

FORTH is a language composed almost entirely of subroutine-like procedures. When a procedure is executed, a return address is pushed onto the return stack and removed as the procedure exits.

In FORTH the procedures are called 'Words' (and what else is a language composed of?), and each Word has a distinct NAME of up to 31 ASCII characters (excluding 'space', CR or LF or NULL).

The collection of Words which compose the language is the DICTIONARY, arranged in the form of a linked list.

When a Word is used, the dictionary is searched from the top (most recent) end to find the required word. What is 'found' is the start address of the Word. Once found, the Word is either executed, if in execution mode, or compiled, if in compile mode. Either way, the start address is all that is required for either of these.

The file FORTH contains around 25Ø Words, some of them machine language primitives, most of them written in FORTH. Yes the language is written in itself.

The whole concept of FORTH programming is to build new Words which consist of a sequence of calls to existing Words. This sequence of calls then takes place when the new Word is executed.

Of course this new Word can then in turn be called by further new Words, thus building up the complexity of what each procedure (Word) can do until finally, one Word invokes your entire application.

Each new Word is linked into the dictionary, such that it is indistinguishable in structure from the rest of the language.

The act of programming literally extends the language (dictionary), to generate a new, extended dictionary of words which can carry out your desired function.

It is possible to save the new extended dictionary as a new version of the language which can be loaded and run directly. That is, you have the capability to create different FORTH's, for specific applications.

The subject of the dictionary and its structure will be covered in more detail later on.

Virtual Memory

ORIC–FORTH implements a very simple disc operating system emulated in RAM, which views the disc as consisting of numbered blocks, each block being 1K bytes regardless of actual disc sector size. The blocks are numbered from Ø up to the capacity of the disc.

If the user wishes to access block 'n', a simple operation brings that block into a buffer in RAM memory, where it can be manipulated. If the block is altered in any way, the updated version will be rewritten to disc automatically, (if it has been altered).

This simple scheme means that the whole of the disc can be 'addressed' as though it were memory, the address consisting of two parts:

The Block number
The byte address (Ø to 1Ø23) within the block

Chapter 3

Getting Going – Some Simple Examples

Assuming you have loaded FORTH, and got the 'OK' prompt, then first type EMPTY–BUFFERS <CR> to initialise the I/O buffers. ALWAYS DO THIS after a cold start unless you have some specific reason for not doing so.

1. Typing in response to 'OK'.

   The system is in terminal input mode, and is waiting for keyboard input as indicated by the presence of the cursor. Anything you type in is initially stored in a Terminal Input Buffer (TIB), 80 characters long. Nothing really happens until you type carriage return <CR>. If you try to enter more than 80 characters, the input routine 'closes' the input with a <CR> for you. After typing <CR>, the FORTH outer interpreter starts to work its way along the input line (as stored).

   The interpreter looks for groups of characters, **separated by spaces:**
   *If it finds a FORTH word, that word is 'executed'.
   *If it finds a number, that number is pushed on to the stack.
   *If it cannot recognise what you have entered, it aborts back to input mode, displaying a '?' and the thing which it did not like.

2. **Simple Arithmetic**

   If you enter 2 3 + . <CR> then what you get is 5 OK What happened? The '2' and '3' were pushed on to the stack. '+' means 'add the two top stack entries together, and leave the answer on top'. Finally '.' means 'print the top of stack as a number'.

   Try: 4 5 + 6 7 + * . Should give 117 OK

3. **Number Bases**

   Forth can work in different number bases, and can change at any time - you can use it as an OCTAL/DECIMAL/HEX/BINARY calculator. At cold start, FORTH starts in DECIMAL. Typing HEX <CR> changes it to a hexadecimal machine. DECIMAL <CR> changes it back again.

In the following, **bold type** shows what you typed (terminated by <CR>); FORTH always finishes with OK to show it has finished the current set of 'commands' and is ready for more.

HEX OK	
3BE8 C8 + . 3CBO OK	(hex addition)
25 2F * . 6CB OK	(hex multiplication)
DECIMAL 1348 HEX . 544 OK	(decimal to hex)

Base changes occur by storing the relevant value in variable BASE, so

8 BASE ! OK	(stores 8 which means OCTAL)
6 3 * . 22 OK	(octal multiplication)
22 DECIMAL . 18 OK	(octal back to decimal)

4. Adding a New Word to the Dictionary

So far everything you have typed has been executed immediately after typing <CR>. In order to add a new word to the dictionary, FORTH must change to COMPILE mode so that the right things are compiled into the dictionary list rather than executed.

Suppose we wish to create a word which takes a number from the top of the stack and returns the CUBE of that number, we can try this from the terminal first.

2 DUP DUP * * . 8 OK

Explanation: '2' goes on the stack, DUP DUP makes two extra copies (3 altogether), * * multiplies all these together, leaving the CUBE on the stack, for '.' to print. To make this a part of the dictionary, type as follows:
: **CUBE DUP DUP** * * ; <CR>OK and then
5 CUBE . 125 OK

If you type VLIST (CR) and Control C, you will see that CUBE is now at the top of the dictionary, and can be used like any other FORTH word. This is the result of the colon .... semicolon pair which define a new Word to be compiled.

: abcd means 'this is a new Word called abcd'. The Forth words which then follow are compiled into the dictionary under the name 'abcd'. Finally the ';' means 'this is the end of the new Word'. Remembering that 8 BASE ! sets OCTAL number base, we could now go

: OCTAL 8 BASE ! ; OK to define an operator which will set OCTAL number base if you type OCTAL < CR >. Similarly you could have

: BINARY 2 BASE ! ; OK

5. **A DO ..... LOOP**

This is a simple loop with a counting index, (a bit like a FOR ...NEXT loop in BASIC).

DO takes two variables from the stack; the initial value of the loop counter is on top, and the final value +1 is next one down on the stack.

Example: ('I' returns the value of the loop counter)

DECIMAL OK

: 10–CUBES	(PRINT A TABLE OF CUBES Ø TO 9)
1Ø Ø DO	(set up the loop end and start)
CR I . I CUBE .	(print a number and its cube)
LOOP CR	(end of the loop, print a carriage return)
; OK	(end of new WORD)

now we can execute the new word

1Ø–CUBES <CR >

Ø	Ø
1	1
2	8
3	27
4	64
5	125
6	216
7	343
8	512
9	729

6. **The IF ..... ELSE ..... ENDIF conditional (or IF ..... ELSE ..... THEN)**

'IF' looks at the top-of-stack **(and removes it).**

It interprets this value to be either false (= Ø) or true (non-zero), and executes the appropriate part of the conditional statement as follows:

IF .....execute this part if true .....ENDIF otherwise come to here
↑
test value here
↓
IF ..... true part .....ELSE ..... false part ENDIF continue here
(note 'THEN' is an alias for ENDIF)

So here is an example which returns the absolute value of the top-of-stack number. Note '**Ø <**' is a test of top-of-stack which leaves a 'true' if the top-of-stack is less than zero i.e. negative.

```
: ABS–VALUE
DUP Ø <                 (copy the number, test its sign)
IF MINUS ENDIF          (change sign if negative)
;                       (end of this word)
```

and then try it

```
1Ø ABS –VALUE  . 1Ø OK
–5 ABS–VALUE  .  5 OK
```

7. The BEGIN ..... UNTIL LOOP

This loop takes a truth value as its argument, usually computed within the loop, which is tested by UNTIL . If this is false, the program loops back to BEGIN . If it is true, the program continues past the UNTIL to the following instruction.

Example

```
: 1ØCUBES               (name of this word)
Ø                       (initial count value)
BEGIN                   (start of loop)
CR DUP . DUP CUBE .     (print a number and its cube)
1 +                     (increment the index)
DUP 1Ø =                (test for index = 1Ø)
UNTIL                   (end of loop, exit if true)
CR DROP                 (throw away final index)
; OK                    (end of word)

1ØCUBES                 now execute it

Ø     Ø
1     1
2     8
3     27
4     64
5     125
6     216
7     343
8     512
9     729
OK
```

8 TEXT INPUT and OUTPUT

Outputting text strings will generally use the word TYPE for internally generated strings, or . " FRED" to generate 'fred', which was a string literal. Subsidiary operators for text output include;
–TRAILING
EMIT
SPACE
SPACES
and ERASE, FILL, BLANKS, for presetting string storage areas.

Note that in FORTH, all strings are stored with their length in the first byte – so maximum length is 255 characters.

Inputting text streams makes use of QUERY, EXPECT, and the EDITOR word TEXT which moves input strings to the buffer area which starts at PAD.

Comparison of strings can be done using the Editor words – TEXT and MATCH.

9. NUMBER INPUT/OUTPUT

All number I/O takes place in the current BASE, so ensure this is correct when programming number I/O.

The principle numeric input word is called NUMBER, which takes a string of characters at a given address and tries to convert them to a double precision integer.

When you type 123 <CR>, it is the NUMBER word which converts the string "1" "2" "3" to binary and puts it on the stack. Note that 123 <CR> generates a single-precision (16 bit) number.

If you type 123. <CR>, the NUMBER routine recognises the decimal point as a request for this to be double precision (32 bit) integer.

Note that 1.23 will also be converted as 123, but variable DPL will hold 2 to indicate 2 decimal places were found on the input conversion.

In order to make number input a bit easier, a new word IN# exists on cassette extension-screen 1, which does all the necessary things to get single precision numbers from the keyboard and put the result on the stack.

Number Output

To output a number, it has to be turned into a string. The operators '.' '.R' 'D.' and 'D.R' do this for you for normal output. These use formatting/conversion operators which are available to you for special conversions. These operators are;

<# #S # HOLD SIGN # >

* You should note that output number conversion takes place RIGHT TO LEFT.
* These primitives always work on double-precision input values.
* The string for printing is generated DOWNWARDS from PAD. The following example demonstrates some of the features you can do – it takes a 16 bit integer from the stack and prints it as hours : minutes : seconds

First we need a word which inserts the : character into the string.

```
HEX : ':' 3A HOLD ; DECIMAL
```

This defines the new word ':' which will do the trick. (Hex $3A is the : character)

Next, an operator to convert in units of 60 (for seconds and minutes).

```
: :ØØ # 6 BASE ! # ':' DECIMAL ;
```

So word :ØØ goes as follows:
#converts the least significant digit in base 1Ø
6 BASE ! sets BASE 6
#converts the next digit in base 6
' : ' inserts the : symbol
and finally DECIMAL is restored
and finally

```
: TIME Ø <#:ØØ :ØØ # # #> TYPE SPACE ;
```

TIME expects the value for output on the stack. It adds a Ø to the stack to make a double precision number.

<# indicates "start of number conversion routine"

:ØØ converts the least significant part of this number to a string in base 6Ø (for the seconds)

:ØØ again does the minutes

converts two more digits (in decimal) for the hours

#> says 'end of conversion

TYPE then types the resulting string

So 65 TIME < CR >would print ØØ:Ø1:Ø5

Note that # #S SIGN HOLD can only be used between the <# and #> symbols

Conclusions

By now, if you type VLIST, you will find you have added a few new words to the dictionary. You could do one of two things – you could FORGET them, to free up the space in memory, or, if this was an application program, you could re-set the boot-up parameters to remember the new words as a permanent part of FORTH, and save the whole new dictionary as a new version of the language. (More on this in an appendix).

Chapter 4

Editing and Creating Source Programs

Having seen how you can give commands to FORTH, and create new words, let's now see how to make a proper source program, using the Editor.

First the editor must be loaded from cassette. It exists as screens 1 to 7 on the source tape, therefore having aligned the tape and set SPEED correctly, type 1 7 CLOAD < CR > and play the tape.

The seven screens will now be loaded, and OK will be returned at the end. If you wish to see the source text, then 1 LIST <CR > will display the first 4 lines of screen 1. Any key except ↓ displays the following four lines, ↓ scrolls through to the end.

Also 1 7 INDEX <CR >will display the comment lines at the top of screens 1 to 7.

The source text can now be compiled into the Forth dictionary, which simply requires that you type 1 LOAD <CR >.

The 7K of source text will now be compiled (taking 3Ø – 4Ø seconds) to 1.5K of Forth object code. Two messages "xxxx ISN'T UNIQUE" will be generated (don't worry) and finally the message "EDITOR LOADED" will appear.

To use the Editor, now type EDITOR < CR > . You now have to choose a suitable block to put your new program – lets say you choose block 4. To clear this of any rubbish, you can type:-
4 CLEAR <CR > or to see what is there, type
4 LIST < CR > which displays that block (also called a screen) 4 lines at a time (press any key to get subsequent groups of 4 lines, or ↓ to scroll through to the end).
Each 'screen' consists of lines Ø to 15, each of which can hold 64 text characters.

To enter some new text on line Ø, enter the command
Ø NEW <CR > which 'opens' line Ø for text entry. Anything you type up to the next <CR > will be entered into that line.

By convention, line Ø of each screen is a comment line, describing the contents, so try entering:- (THIS IS AN EXAMPLE SCREEN)
<CR >

The editor will now prompt you for line 1. If you wish to leave this line a blank, type a space, <CR> , and line 2 will be prompted. Suppose we enter the CUBE definition from the previous chapter
: CUBE DUP DUP * * ; (n---cube-of-n) <CR>

If that is all you want to enter onto this screen,type <CR>straightaway in answer to the prompt for the next line. You can now try command L to display your new screen, and try out some of the other Editor commands. *

You could now LOAD your new screen, add the words in it to the dictionary.

In the following section, some of the editor commands and their effects are described.
* When you have finished editing a screen, enter the command FLUSH <CR> this ensures that your edited screen is put back onto the disc.

Text Input Commands

Having selected the screen for editing (say screen 4), by going 4 LIST <CR> , or 4 CLEAR <CR> the following commands are available for insertion of text. 1 P This text will go on line 1 <CR>

P means 'Put a new Line', and the proceeding number is the line selected. All the characters after the space after P are put on the line (1 in this case), overwriting anything already present. Max line length is 64 characters. Beware of not putting anything at all. If you accidentally go 1 P <CR> , a 'null' will be put in this line, which will cause an error later. If you do this, go 1 E <CR> to erase the line completely.

n NEW <CR> selects line n for input. The screen is displayed to you, with line numbers, as far as line n, where it stops and prompts for your input. Now type in the required text, finishing off with a <CR> . immediately, closes the input, and the rest of the screen scrolls through.

n UNDER <CR> displays the screen down to the beginning of line n + 1 , and waits for your input, as in NEW.

The original line n + 1, and succeeding lines are moved downwards. Line 15 is lost.

Screen Editing

These commands operate on whole screens.

n LIST <CR> displays screen n
n CLEAR <CR> clears screen n to all spaces
nl n2 COPY <CR> copies screen nl to n2
L re-lists the current screen, and the current cursor line
FLUSH forces all amended screens back onto the disc after editing.

Line Editing

These commands operate on a selected line within the current screen.

Use is made of a buffer area in RAM called the PAD (short for Scratch-pad). PAD is always 68 (decimal) bytes higher in memory than the top of the dictionary.

n H <CR> copies line n into the PAD buffer (Hold)
n D <CR> copy line n into PAD, and delete the line from the screen. Lines n + 1 to 15 are moved up, and new line 15 is cleared.
n T <CR> Type line n on the terminal, and save it in PAD
n R <CR> Replace line n by the line stored in PAD
n I <CR> Inserts the stored line in PAD into line n. Existing lines n to 14 are moved down to make room. Old line 15 is lost.
n E <CR> Erase line n to space.
n S <CR> Spread out at line n. Lines 1 to 14 are moved down, leaving line n clear.

String Editing and Cursor Control

Editing operations on character strings within the current line take place with reference to the editing cursor, displayed as a crosshatch character '■'.

Initially, TOP, sets the edit cursor to the top of the screen. Going back to the example above, where definition CUBE was entered onto line 2, then 2 T <CR> will display line 2, with the edit cursor at the start of the line ■: CUBE DUP DUP * * ;

To find the string DUP, type F DUP <CR>. The screen is searched forward from the editor cursor position to locate a match to the string you have requested, and when found, displays as follows:

: CUBE DUP ■ DUP * * ; Edit Cursor

Going N <CR> will continue the search for the NEXT appearance of the same text

: CUBE DUP DUP■* *;

Executing B <CR> takes the cursor back by the length of the text string located.

: CUBE DUP■DUP * *;

Note that the cursor can also be moved directly by the M command.

Having located the part of the line you wish to operate on, the following commands allow you to delete/change strings of characters.

X DUP <CR> Command X searches for and deletes the string

: CUBE DUP■* *;

C DUP < CR > Command C copies the string that follows into the cursor position

: CUBE DUP DUP■ * * ;

Other commands are TILL text and n DELETE, which are explained in the glossary.

Chapter 5

An Example of Program Development – Simple PRINT Utility

Let us think of a starting specification for this:

"To print a contiguous block of screen numbers, at three screens per page, with page number, title line and system ident message".

Forth is a top-down language – that is, one where problems are best solved by starting from the top of a program, and working inwards, refining at each step.

So, suppose we want to issue a command 'from' 'to' PRINT <CR>, and the spec above says it does 3 screens per page. Presumably if there are less than three, then just those left are printed.

Thus the first attempt might be something like

```
: PRINT  SET–PAGE–1
         MORE–THAN–3–SCREENS?
         IF PRINT–PAGE–FULL ELSE PRINT–WHATS–LEFT
         ENDIF
         REPEAT–TILL ALL DONE ;
```

Of course this won't work, but it has all the essential elements.

If we also define Ø VARIABLE P# for page number, then SET–PAGE–1 becomes 1 P# !

Also, we haven't yet turned the printer on or off!

So for our next attempt, we can have

```
: PRINT 1 P# ! PR–ON (printer on)
SET–LOOP–UP BEGIN (begin loop)
≥3–LEFT–TO–DO? (enough for a whole page?)
IF DO–PAGE ELSE DO–REST ENDIF
UNTIL (till all done)
PR–OFF ; (printer off)
```

and now we can define a few more things.

```
HEX : PR–ON F57B DUP DUP 1BF4 ! 1C22 ! 1CID ! ;
      : PR–OFF CC12 DUP DUP 1BF4 ! 1C22 ! 1CID ! ;
DECIMAL
```

these modify Forths I/O handlers to enable/disable the parallel printer port.

How about DO–PAGE? if this took in the arguments start–screen–number and count–left, and returned the updated versions, i.e. start +3 and count –3, then it automatically leaves the correct arguments to be called again in the main loop.

With a bit of trial and error, I got

```
: DO–PAGE 3 – SWAP 3 + SWAP OVER DUP 3 – PRINT–IT ;
```

This DO–PAGE adjusts the start and count, and also then produces values 'from' 'to' for PRINT–IT, which is going to do the real work.

Similarily, DO–REST should also return the two adjusted values, except that the final 'count-left' will be ZERO.

```
: DO–REST (from to ––– from Ø)
>R Ø OVER DUP R> + SWAP PRINT–IT ;
```

This works out nicely, because the 'count–left' is on the top of the stack, and is only Ø when all the printing is finished, so it can be used to test for exiting from the main print loop.

We also need to alter the 'from' 'to' numbers which are input initially, to the 'from' 'count' required by DO–PAGE and DO–REST. This same 'count' can then be tested to see if it is >2.

So now we have

```
:       PRINT                   (from to–––)
        1 P# !                  (set page 1  )
        PR–ON CR                (printer-on, CR)
        OVER – 1+               (change 'from' 'to' into 'from'
                                 'count')
        BEGIN                   (start print loop)
        DUP 2 >IF               (test count value)
        DO–PAGE ELSE            (full page if >2)
        DO–REST ENDIF           (else the rest)
        CR                      (force output to occur)
        DUP Ø = UNTIL           (loop until Ø count is true)
        DROP DROP CR            (throw away unwanted variables)
        PR–OFF ;                (printer off and done)
```

PRINT–IT is next, and it receives as input the 'from' and 'to' values, which can be used in a DO....LOOP'

```
:       PRINT–IT  (to from ––– print them)
```

```
PR–ON DO I PRINTSCRN LOOP CR 15 MESSAGE CR CR CR
CR PR–OFF ;
```

So this now does the whole or part page, calling PRINTSCRN to print one screen, and the system identification message (number 15) at the bottom.

Now we need PRINTSCRN – this can be borrowed entirely from the LIST function:

```
:       PRINTSCRN  (n ––– print this one)
        DECIMAL CR DUP SCR ! . " SCREEN " . 16 Ø DO
        CR I 3 . R SPACE I SCR @ .LINE LOOP CR ;
```

which gets lines at a time in a DO.....LOOP and calls .LINE to print them.

So this is almost complete now, and the final utility is reproduced at the end of the chapter, using codes for an OKI printer which can do double size characters, See if you can work out how it fetches the print header message and adds it and the page number to the top of each page.

Also reproduced is a sample stack diagram. These are invaluable in trying to visualise what is happening on the stack, and their use is highly recommended.

```
SCREEN 3
Ø       (PRINTING UTILITY 1 of 3                    WANB NOV 81)
1       FORTH DEFINITIONS DECIMAL
2       0 VARIABLE P#
3       (LINES 6 TO 8 ARE PRINTER DEPENDANT)
4       HEX  :  PR–ON F57B DUP DUP 1BF4 ! 1C22 ! 1C1D ! ;
5    :  PR–OFF CC12 DUP DUP 1BF4 ! 1C22 ! 1C1D ! ;
6       :BIGCH 1F EMIT ;
7       :  NOMCH 1E EMIT ;
8       :  TOF  CR  .  " READY?" KEY DROP ;
9       DECIMAL
1Ø   :  PRT–SCREEN                          (n ––– prints scrn n)
11      DECIMAL CR DUP SCR ! ." SCREEN"  .  16 0 DO
12      CR I 3 . R SPACE I SCR @ .LINE LOOP CR ;
13→
14
15
```

```
SCREEN 4
Ø       (PRINTING UTILITY 2 of 3                    WANB NOV 81)
1
2    :  PRTHED (output page header)
3       PAD C@ 38 MIN 1 MAX PAD C!  BIGCH SPACE PAD COUNT
        TYPE 31
4       PAD C@ – DUP Ø < IF CR DROP 32 ENDIF SPACES
5       ." PAGE" P# @ 3 .R CR CR 1 P#+! NOMCH ;
6
7    :  PRINT–IT  (endstart ––– print these scrns)
8       PR–ON PRTHED DO I PRT–SCREEN LOOP
9       CR 15 MESSAGE CR CR CR PR–OFF ;
1Ø
11   :  DO–PAGE  (from count ––– frm+3 count–3  do 3 scrns)
12      3 –  SWAP 3 + SWAP OVER DUP 3  –  PRINT–IT ;
13→
14
15
```

```
SCREEN 5
Ø       (PRINTING UTILITY 3 of 3                    WANB NOV 81)
1
2    :  DO–REST (from to ––– from Ø do 1 or 2 scrns left)
3       > R Ø OVER DUP R > + SWAP PRINT–IT ;
4
5
6
7    :  PRINT                 (n m ––– print screens n to m, 3 per page)
8       CR  ." HEADER:" EDITOR ENTER
```

```
9       1 P# ! PR–ON CR OVER – 1+ BEGIN
10      DUP 2 > IF DO–PAGE ELSE DO–REST ENDIF
11      DUP Ø = TOF UNTIL
12      DROP DROP PR–ON CR CR CR CR PR–OFF ;
13  ;   S
14
15
```

ORIC FIG–FORTH

APPLICATION : PRINT UTILITY
WORD : DO–PAGE

STACK				TOP
			From	Count
		From	Count	3
			From	Count–3
			Count–3	From
		Count–3	From	3
			Count–3	From+3
			From+3	Count–3
		From+3	Count–3	From+3
	From+3	Count–3	From+3	From+3
From+3	Count–3	From+3	From+3	3
	From+3	Count–3	From+3	From
			From+3	Count–3

WORDS

```
        (enters with 'from' and 'count' on the stack)
: DO–PAGE
  3
  –
  SWAP
  3
  +
  SWAP
  OVER
  DUP
  3
  –
  PRINT–IT
  ;
```

NOTES: DO–PAGE

Enters with the starting screen'from' and 'count' left. Since a PAGE is 3 screens, it generates the updated 'from' and 'count', and the requisite arguments for PRINT–IT, in the form 'from+3' and 'from' i.e. these are 'end' and 'start' screen numbers for a PAGE of 3 screens.

Chapter 6

FORTH Dictionary Structure

Since 99% of FORTH is in the dictionary, it is very worthwhile to investigate its structure, which we can do by reference to our favourite example, the CUBE command. When you typed in

```
: CUBE DUP DUP * * ; <CR>
```

the new word CUBE was added to the dictionary. In memory, it actually looks as follows, where each rectangle represents a byte of memory.

$84	NAME FIELD ADDRESS	Dictionary Header for word 'CUBE'
C		
U		
B		
E + $8Ø	----------	
low link	LINK FIELD ADDRESS	
hi link	----------	
low address of	CODE FIELD ADDRESS	
hi DO-COLON		
low address of	PARAMETER FIELD ADDRESS	Parameter BLOCK of 'CUBE', giving the list of operations to be performed.
hi DUP		
lo address of		
hi DUP		
lo address of		
hi *		
lo address of		
hi *		
lo address of		
hi SEMI-COLON	----------	
	NEXT FREE BYTE	

Higher Memory

As you see, a dictionary entry comprises a HEADER SECTION, and a PARAMETER section. The first contains all the necessary information that describes the name of the entry and its type; the second part contains a list of addresses which effectively point to those words which make up the new word.

The header block is subdivided as follows:

Name Field: This starts with a length byte, whose five LSB's indicate the length of the following ASCII string, which is the name of the FORTH word. The MSB of the length byte is also set to identify it. Then comes the ASCII string of the name, and the final character also has its MSB set, to mark it.

The address of the length byte is generally known as the Name Field Address, or NFA.

Link Field: This contains the Name Field Address of the previous dictionary entry. Thus these Link addresses chain right through the dictionary, allowing it to be searched from the most recent end downwards. The address of this field is the LFA.

Code Field: This is the field which defines the 'type' of word, and it's address is called the Code Field Address (CFA). The CFA of the definition is also the address at which execution of a word starts; the contents of the CFA being the ADDRESS OF REAL, EXECUTABLE, MACHINE CODE.

In this example the code field contains the address of DO–COLON, a code routine to perform a FORTH subroutine type of call, appropriate to a COLON definition such as CUBE.

Any Word compiled by ':' will have the address of DO–COLON in the CFA, other Word types will have other addresses in here, depending on the 'class', or 'type' of Word.

The Parameter Field contains, in this example, a list of the Code Field Addresses of the Words which make up 'CUBE', and the first address is called the PFA. For COLON definitions, the list terminates with the address of SEMI–COLON, which is effectively an 'end-of-subroutine' function – a sort of FORTH equivalent of RTS in assembler. The contents of the parameter field will also vary with the 'type'of Word.

Vocabularies

The link addresses chain all the dictionary Words into a long list, in the first instance, is the whole of the FORTH VOCABULARY. For convenience, and searching speed, you can segregate new Words into different Vocabularies, an example being the EDITOR.

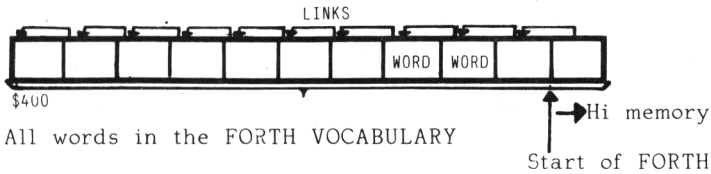
Vocabularies have two main effects: First they increase compilation speed, by allowing dictionary searches to start in the right 'area'. Secondly, you can have Words with the same Name in separate

Vocabularies, with less risk of confusion (e.g. Editor 'R' command and FORTH 'R').

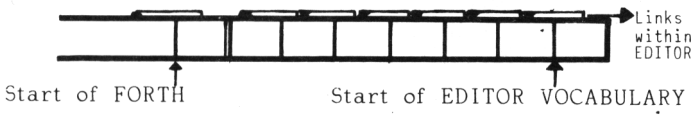
When a new Vocabulary is set up, the Link address chaining is modified to ensure that new Words are compiled into the CURRENT VOCABULARY.

An Example

1. After loading FORTH, the initial dictionary structure is as shown:

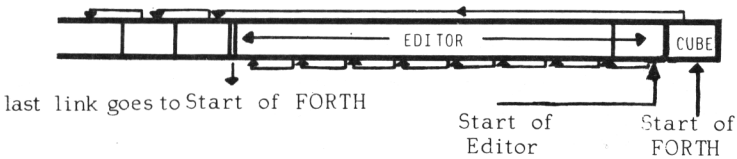


2. On adding the EDITOR, a second Vocabulary exists:



When you type EDITOR <CR>, this tells the interpreter that all dictionary searches will start at the top of the Editor Vocabulary. Typing FORTH <CR> resets the pointer, called the CONTEXT VOCABULARY pointer, to the top of FORTH, so the Editor is skipped over.

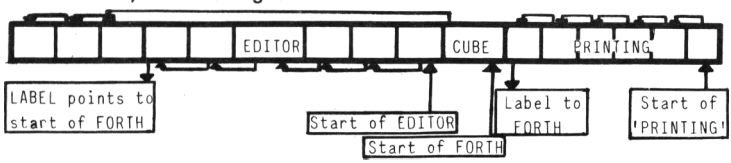
3. Now suppose we add : CUBE to the FORTH vocabulary.



Notice how CUBE is added to the end of the dictionary, but the FORTH linkage now skips right over EDITOR.

As before, the end of EDITOR points through to the start of FORTH; the general rule being that all vocabularies fall into the main FORTH vocabulary, but not into each other.

4. If we now added, say, some new Words into a Vocabulary called PRINTING, we would get:



Now there are three VOCABULARIES: FORTH, EDITOR, and PRINTING.

FOR SAFETY, ALL NEW VOCABULARIES CHAIN (i.e. LINK) TO FORTH, NEVER TO EACH OTHER.

5. To set up a Vocabulary, the relevant instruction is FORTH DEFINITIONS (set FORTH as main VOCABULARY) VOCABULARY FRED IMMEDIATE (declare a new VOCABULARY called FRED) then FRED DEFINITIONS sets FRED as the CURRENT vocabulary (i.e. new Words are added to the FRED list).

   Finally, FORTH DEFINITIONS goes back to FORTH at the end of FRED's additions.

   To use Words that are in FRED, type FRED <CR>.

The Other 1%

We said that 99% of FORTH is in the dictionary. The 1% that is not (apart from the stacks), is the CASSETTE–BUFFER AREA, and the USER VARIABLE BLOCK.

These are located at the top of memory (see the memory map).

The 'user variables ' is actually a block of variables used by the system, though they are all accessible to you. They are called USER variables because their values are particular to a specific user. FORTH can be made multi-user, and in such cases, there would be a block of USER variables for each person and only the user pointer then needs to be changed to reference each.

The actual contents, their meanings, and the relevant boot-up values, are given in an appendix, and the glossary.

The cassette-buffer is the area in which you manipulate cassette information. As supplied, there is 1 buffer, 1028 bytes long, used in turn by the FORTH cassette manager. The buffer is composed as follows:

2 bytes:	Holds the 'disc' - block - number which is loaded into this buffer.
1024 bytes:	The data area, holding a 1K 'disc' block.
2 bytes	Contain Ø, to make the end of the buffer.

If you wish to get at 'disc' information, the relevant FORTH words are BLOCK, BUFFER, UPDATE and FLUSH. See the Glossary.

Chapter 7

The Code Field

In every FORTH dictionary word there is a two byte location called the CODE FIELD. This very important field determines the TYPE of word. and how it executes.

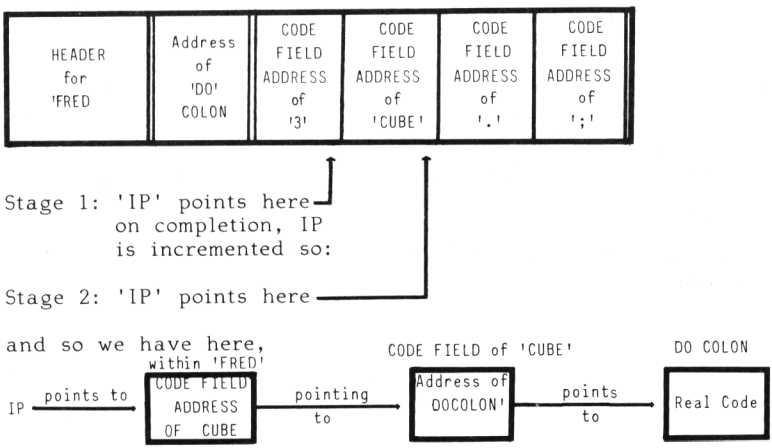
The contents of the CODE FIELD are the address of a real machine language routine to be executed when the word is first called, i.e. the CODE FIELD ADDRESS represents the start of the word.

To see how it functions, we must consider in more depth how FORTH works.

As described in Chapter 7, most FORTH words consist of a list of addresses of other words to be executed. The FORTH program counter, called IP (Interpretive Pointer), follows this list item by item, acting in a very similar way to the processors' own program counter.

Let us first look at the ':' type of word, the most common, and our example CUBE. Suppose we have another word which has somewhere in it the word CUBE, e.g. : FRED 3 CUBE . ;

When this word has started, IP points first to '3', which is actually defined as a FORTH word to put the value '3' onto the stack. IP is then incremented by two, and points to the list entry for CUBE, which contains the CODE FIELD ADDRESS of 'CUBE'. FORTH now performs an INDIRECT JUMP instruction. This means it ends up not at 'CUBE', itself, but at the machine code routine pointed to by 'CUBE's Code Field. That sounds very confusing. Lets try and draw it.



Here are the two levels of indirection for which FORTH is renowned - i.e. two levels of pointing to the real code.

The Jump Indirect (which is part of the FORTH inner interpreter) takes execution straight to 'DOCOLON'.

'DOCOLON' behaves like a subroutine call, in that it says:

'In order to start executing the new word (CUBE in this case), IP must be changed to point to the list within CUBE – its parameter field. To do this we must first save where IP is at the moment'.

And that is what DOCOLON does. It first saves IP on the processor RETURN stack, and then loads IP with the address of the start of 'CUBE's list of addresses.

CUBE finishes with a ';'. This is the reverse of DOCOLON – it pulls the stored value of IP off the return stack and restores it.

In summary the, DOCOLON is a sort of FORTH 'JSR', and ';' is the equivalent of RTS.

DOCOLON is only one of many things that can be in the Code Field. DOCOLON is only appropriate to words defined with the ':' symbol, as only this means that the new word will consist of a list of old words, hence if the CODE FIELD of a word contains the address of DOCOLON, the word is a 'colon definition'.

Other Word Types

The other common word types in FORTH are CONSTANT, VARIABLE, USER and CODE.

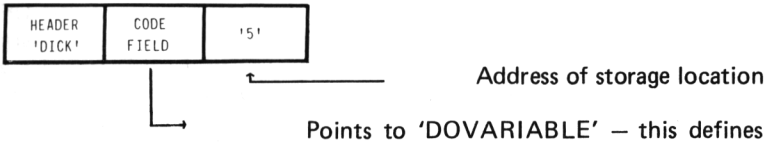
Example 1

3 CONSTANT FRED defines the word 'FRED' which is a constant type, and the value of the constant is 3. Its dictionary entry looks like

Points to DOCONSTANT' – this defines the operation of a CONSTANT, and is a routine which extracts the value '3' and pushes it onto the stack when FRED executes.

Example 2

5 VARIABLE DICK defines a VARIABLE type, with the initial value of the variable set of 5.



Points to 'DOVARIABLE' – this defines the operation of VARIABLE, which is that when DICK executes, the ADDRESS of the storage location is pushed onto the stack.

Example 3

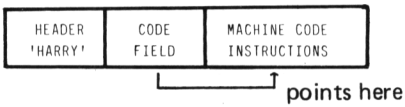
26 USER TOM defines a USER variable, named TOM. User variables are stored in a special block in high memory, and this would define TOM as being the 26th entry in the user block, and when TOM executes, the ADDRESS of the storage location is pushed onto the stack.

Example 4 – with an Assembler

CODE HARRY code

machine code.

defines a machine code routine. In this case, the word contains the code to be executed directly, so the code field points to the start:



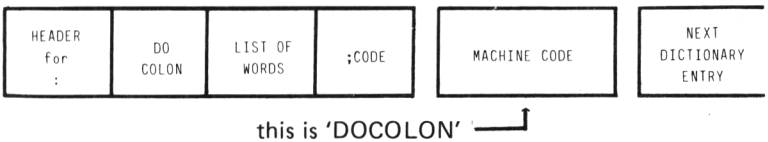
(See also the next chapter)

The next question is – where do the routines DOCOLON, DOVARIABLE etc, actually reside?

The answer is they form part of the relevant DEFINING WORD. A defining word is one of the group of ':', VARIABLE, etc so called because they DEFINE a new word of the appropriate type.

DEFINING words can be thought of as special words, which have two distinct parts, one which specifies how to compile an entry of the correct type, and the second part which defines how it will execute.

For example ' : ' looks like this:



When ':' executes (e.g. when you enter : FRED ..... ;), the 'LIST OF WORDS' are the FORTH words necessary to CREATE a dictionary header whose name is FRED, and whose CODE FIELD contains a pointer to 'DOCOLON', the machine code which follows the ';CODE' word.

Thus a defining word consists of:

* A 'BUILD' part, which BUILDS the correct type of dictionary entry.
* A 'DOING' part, which is the common execution code for words of this type.

Now we get to the best (or worst) part of all!

One of FORTH's important features is the ability to create new defining words. This is something that is very useful, and is a feature found in only one or two other computer languages – of which BASIC is not one.

Two sorts of defining words can be generated – ones where the DOING part is in assembly code, and ones where the DOING part is in FORTH. The first sort offer a faster execution speed, but it is very important to know what you are doing. The second sort are easier to do.

We will restrict ourselves to a simple example, a definition of CONSTANT.

```
: CONSTANT CREATE SMUDGE , ;CODE LDY #$2 < – This is
                                 'DOCONSTANT'
                                 LDA (W), Y
                                 PHA
                                 I NY
                                 LDA (W), Y
                                 JMP PUSH
```

So, : CONSTANT defines the name of this operation.
CREATE SMUDGE , generates the dictionary header for the new word.
; CODE is the clever one which puts the 'DOCONSTANT' start address of the machine code into the code field address of the new word.

This short piece of machine code (for 6502) is what actually gets the value out of a CONSTANT parameter field and puts it on the stack.

The second sort of defining word has the DOING part in FORTH, and is most often used to create types of data structures.

Example: Here is 'CONSTANT' implemented this way:

```
: CONSTANT <BUILDS , DOES> @ ;
```

Unpicking this .. : CONSTANT is standard.

<BUILDS means "everything that follows here up to DOES> is the building part of this defining word". <BUILDS itself takes care of generating FORTH headers.

', ' Means 'takes the word on top of the stack and compile it into the next dictionary location. Remember that to use CONSTANT, a value is supplied first, which will go on the stack, and sits there until ', ' uses it.

So <BUILDS , generates the header part of the new dictionary entry, and ', ' puts the value into its **parameter field.**

DOES> means "everything that follows is the FORTH to be executed whenever the newly BUILT word executed". DOES> also pushes the **parameter field address** onto the stack when the new word executes. Since, in this case, our constant is in the parameter field, '@' gets the value stored at this address.

So if you went 120 CONSTANT MINE, using this definition of CONSTANT, it would BUILD a dictionary header named 'MINE' and embed the value in the parameter field.

When you execute 'MINE', the DOES part is called to get the value and push it onto the stack. Simple really!

Now you work out this one, which creates one-dimensional byte arrays.

```
: BYTEARRAY <BUILDS ALLOT DOES> + ;
```

and would be used 23 BYTEARRAY FRED to make FRED, an array of 23 bytes numbered Ø to 22.

Chapter 8

Machine Code Words

It may be, that in order to improve execution speed, or to link to routines in EPROM, that a machine code routine is required.

The true method is to use a structured FORTH assembler. However, it would be useful to see how machine code can be written directly in FORTH without the use of an assembler. The instructions for the assembler supplied with your cassette are described later in this manual.

The method used is to supply the assembler code as hex words or bytes, and to use the FORTH words comma ',' and 'C,' which places the bytes into the dictionary.

Let us have a simple example. To create the 'ZAP' sound:

In assembler, the actual code required would be:

```
STX XSAVE  (save X)
JSR $F41B  (ZAP)
LDX XSAVE  (GET X)
JMP NEXT
```

Where NEXT is the FORTH linkage address to which ALL machine code routines must go when they exit.

So first get the Hex codes for these instructions:

So you would get:

Address	Hexcode	Instruction
4000	86B5	STX XSAVE
4002	201BF4	JSR ZAP
4005	A6B5	LDX XSAVE
4007	4C4404	JMP NEXT

Step 2

Write down the Hexcodes, in pairs, in the order given: 86B5 201B F4A6 B54C 4404

Step 3

Reverse the bytes in each pair: B586 1B20 A6F4 4CB5 Ø444

We can now make a compilable machine code routine.

```
HEX
CREATE ZAP B586 , 1B20 , A6F4 , 4CB5 , 0444 , SMUDGE
```

CREATE makes the dictionary header.
SMUDGE terminates the routine.
HEX is necessary because the code is in HEX.

(This method of "hex-code, hex-code," will also work with ;CODE for making new defining words).

In order to make much use of an assembly code routine, you need to know some more details of what you can and can not do.

*** At the start of a Machine-code Routine**

On entry to a routine, the 6502 Accumulator and Y-Register are available for use. Y is always set to zero on entry.

The X register is the FORTH stack pointer and should not be used, or if it is, save it first in the location XSAVE (which is HEX B5), and restore it again at the end.

*** Stack Access**

The FORTH stack deals with 16-bit numbers. The current top-of-stack location is accessed as address Ø,X (lo-byte) and 1,X (hi-byte). The next item on the stack would be at 2,X and 3,X (e.g. as in LDA Ø,X).

If you wish to make room for a new item on the stack, then DEX DEX is required. Similarly, INX INX winds the stack pointer (X) past the top entry to DROP it.

*** Zero—Page**

All of zero-page locations B6 to FF are available for your use.

*** Branches**

DO NOT USE 'JMP' instructions except as given in the next section. If you want an unconditional branch, fiddle it by using

```
CLC
BCC FRED
```

which uses no more space and is relocatable.

*** Exit Points**

A variety of these exist, and they all eventually return to 'NEXT', doing some commonly used functions on the way. They **must** all be invoked with a JMP instruction.

NAME	HEX ADDRESS	FUNCTION
NEXT	Ø444	Proceeds to the next FORTH instruction.
POP	Ø5EE	Pops (removes) the top stack entry and goes to NEXT.
POPTWO	Ø5EC	Pops the two top stack entries and goes to next.
PUSH	Ø43D	Creates a new stack entry, and puts into it the high byte from the accumulator, and the low byte from the 6502 return stack (pushed onto it before jumping to PUSH).
PUSHØA	Ø7DC	Similar to PUSH, except that the high byte of the new entry is automatically set to ZERO, and the low byte is the current Accumulator value.
PUT	Ø43F	Similar to PUSH, except that the new stack entry overwrites the current top-of-stack item.

*** Example**

This example is taken from the FORTH interpreter, and is the code which comes out of the '+' operation, to add the two top-stack items together.

```
     18          CLC        ;  Carry = Ø
B5   ØØ          LDA  Ø,X   ;  Add low bytes
75   Ø2          ADC  2,X   ;  Add low bytes
95   Ø2          STA  2,X   ;  And store
B5   Ø1          LDA  3,X   ;  Add high bytes
75   Ø3          ADC  3,X   ;  Add high bytes
95   Ø3          STA  3,X   ;  And store
4C   EE   Ø5     JMP  POP   ;  Exit dropping old top item.
```

so writing this out gives:

18B5 ØØ75 Ø295 Ø2B5 Ø175 Ø395 Ø34C EEØ5

and swapping the byte pairs and adding the other bits then gives us

```
HEX
CREATE +  B518 , 75ØØ , 95Ø2 , B5Ø2 ,
          75Ø1 , 95Ø3 , 4CØ3 , Ø5EE ,

SMUDGE
DECIMAL
```

Appendix A

Error Messages

Note: Setting WARNING to −1 will cause any error to ABORT and restart FORTH as if from a warm start.

In the following table, 'XXXX' indicates a Word or Words in error.

1. **XXX ?**

   This is issued by

   a) The compiler/interpreter if it can't find Word XXXX in the dictionary. Usually caused by mistyping!

   b) The Editor find routine if it can't locate the requested string.

2. **? Empty Stack** (Message No: 1)

   An attempt was made to POP something from the parameter stack when it was empty.

3. **? XXXX Isn't Unique** (No: 4)

   This is a warning that you have compiled a Word with a name that has been used already. The compilation still takes place.

4. **? Disc Range** (No. 6)

   You have used a screen number that is out of range.

5. **? Full Stack** (No: 7)

   Self explanatory.

6. **? XXXX Compilation Only** (No: 17)

   You have tried to execute XXXX directly from the terminal. It is only valid within a definition i.e. when compiling into the dictionary.

7. **? XXXX Execution Only** (No: 18)

   Similar to above. XXXX is only valid for execution and can't be compiled.

8. **? XXXX Conditionals not Paired** (No: 19)

You have finished a definition without correctly pairing up one or other of the following:

DO	LOOP	
DO	+LOOP	
IF	ELSE	ENDIF
BEGIN	UNTIL	
BEGIN	AGAIN	
BEGIN	WHILE	REPEAT

9. **? XXXX Definition not Finished** (No: 20)

An alternative error message generated by an improperly completed definition.

10. **? XXXX In Protected Dictionary** (No: 21)

You have tried to FORGET a definition which is in the protected area below the FENCE trap address.

11. **? XXXX Loading Only** (No: 22)

Similar to 6 and 7 for instructions that are only valid when loading from disc.

12. **? XXXX Off Current Screen** (No: 23)

The Editor Cursor position has been moved off the current 1K Screen. TOP resets to position Ø.

13. **? Dictionary Full** (No: 2)

The word name displayed is likely to overflow the remaining dictionary space.

14. **? Disc Error** (No: 8)

Some form of loading error.

Appendix B

Variables Stored in 'User Area'

Offset from base of User Area

HEX	DECIMAL	NAME	FUNCTION	LOADED FROM BOOT TABLE
Ø	Ø	(NTOP)		$4ØC
2	2	(BSCH)		$4ØE
4	4	(UAREA)		$41Ø
6	6	(SØ)	Initial parameter stack pointer	$412
8	8	(RØ)	Stores initial return stack pointer	$414
A	1Ø	TIB		$416
C	12	WIDTH	WARM ↓	$418
E	14	WARNING	Described in the glossary	$41A
1Ø	16	FENCE	COLD	$41C
12	18	DP	↓	$41E
14	2Ø	VOC–LINK		$42Ø
16	22	BLK		
18	24	IN		
1A	26	OUT		
1C	28	SCR		
1E	3Ø	OFFSET		
2Ø	32	CONTEXT		
22	34	CURRENT		
24	36	STATE		
26	38	BASE		
28	4Ø	DPL		
2A	42	FLD		
2C	44	CSP		
2E	46	R#		
3Ø	48	HLD		
32	5Ø	Free		
↓	↓			
7E	126			

WARM means warm start reloads the User parameters down to this level.

COLD also resets down to this level.

Appendix C

Saving an Application

When you have developed some application, you might wish to save the compiled form as a run time file. To do this, some of the boot-up table needs to be amended such that a cold start includes your new application.

An example of what needs to be done is at the end of the Editor code, which is reproduced here with comments.

```
FORTH DEFINITIONS DECIMAL
LATEST 12 +ORIGIN    !  (set bootstrap pointer to top of dictionary)
HERE   28 +ORIGIN    !  (sets FENCE to top of dictionary)
HERE   30 +ORIGIN    !  (sets bootup dictionary pointer)
HERE FENCE !            (sets current fence)
```

If you have declared any new vocabularies, you also need to go

```
' XXXXX 6 + 32 +ORIGIN ! (sets initial vocabulary link)
```

Where XXXXX is the name of the most recent vocabulary. Having tacked this lot onto the end of your program, compile it onto FORTH, and then go

```
HEX HERE 1 - Ø D . <CR>
```

and this prints the top address of the whole thing (in HEX)

Loading and running this new file will bring up FORTH plus your application straight away.

One thing that is commonly done is to make a combined FORTH + EDITOR file, to save compiling EDITOR everytime.

FORTH–SAVE will save on cassette your new version of Forth. It will ask for the file name and tell you to press space when ready. Remember to set SPEED to Ø or –1 for fast or slow save. HEX 101 63 ! will make Forth autoload.

TOP OF MEMORY		48K
7K CASSETTE SOURCE BUFFER		97FF 7CØØ
'IK' BUFFER	DAREA	775Ø
'USER'VARIABLES	UAREA	776Ø
	PAD = HERE+68	
↑	HERE	
USER PROGRAMS		
FORTH SYSTEM (DICTIONARY)	'FORTH' 400	
System Variables	3FF 200	
↓ TERMINAL INPUT BUFFER ↑	1FF 100	
System Variables		
FORTH REGISTERS	B5	
PARAMETER STACK	9F 2Ø	
System Variables	1F Ø	

ORIC-FORTH MEMORY MAP

Appendix D

CONTENTS OF CASSETTE

Side 1:	FORTH	
	EDITOR	SCRNØ1–Ø7
Side 2:	ASSEMBLER	SCRNØ1–Ø5
	EXTENSIONS	SCRNØ1–Ø7
	TUNESMITH	SCRNØ1–Ø4 (MUSIC PLAYING DEMO)

Oric Forth on Cassette

Extensions to Standard Vocabulary

Cassette I/O

SPEED User variable for cassette speed: Ø SPEED ! sets fast
–1 SPEED ! sets CUTS (slow)

CLOAD	n	m	---	loads 1K source blocks n to m inclusive into buffers
CSAVE	n	m	---	saves ditto

n must be or = to m

Cassette Primitives

(STORE)	n	---		executes m/c code save routine linking to BASIC
(RECALL)	n	---		ditto loading
SETUP	n	---	n	computes block addresses from block no.
NAME	n	---		sends filename to buffer area

Other Extensions

PAPER	n	---	n=0 to 7 changes PAPER colour
INK	n	---	as for paper
CAPS		---	toggles caps lock
KLCK		---	toggles key clicks
FORTH–SAVE		---	Save Forth on cassette
TEXT		---	Switches to Text mode
HIRES		---	Switches to HIRES mode

APPENDIX E

FORTH Assembler for ORIC–1

Fancy a full macro-assembler for the 65Ø2, with pseudo-structure commands, symbols, equates, operating in reverse Polish (!) with a source code just 80 lines long?

Look no further, here it is.

Featuring:

1. Macro instructions (IF, BEGIN, etc), user extensible as required.
2. Literals in any numeric base (alterable).
3. Expressions using any resident computation algorithms.
4. Nested control structures.
5. Symbol equates.
6. Labels (if desperately required).
7. Assembler source code in a portable high level language.

The Assembly Process

A FORTH code assembly consists of interpreting in the Assembler Vocabulary.

The FORTH outer interpreter tries to match the incoming text stream against the context vocabulary, set to Assembler, defaulting to FORTH in the usual way if there is no match.

Words in the Assembler Vocabulary will specify operands, address modes, opcodes etc, which will be acted upon and at the end of a CODE definition, the new Word is "unsmudged" if no errors are detected.

During the assembly process, each assembler Word will execute as it is encountered. Its function at this instant is "assembling" e.g. generation of an op-code, or an address, Later on, at RUN time, the assembled machine code executes.

Opcodes

All mnemonics (and Macro's) end in a comma " , " in this assembler. The significance of this is:

a) The comma ends a group of assembler words which would correspond to one line of "ordinary" assembly code.

b) The FORTH comma operator " , " compiles a word into the dictionary, thus a comma on the mnemonic reminds that this is the point at which something gets entered into the dictionary.

c) The use of " , " allows you to distinguish between a mnemonic and other words.

(e.g. the HEX number ADC and the add mnemonic ADC,)

An Example

```
CODE      NOT–MUCH
NOP
NEXT JMP,
;C
```

CODE creates a dictionary header named NOT–MUCH and sets CONTEXT to Assembler. This header will direct FORTH to execute the ensuing machine code instructions at run time. NOP, is mnemonic, and the interpreter executed NOP, which compiles the HEX byte SEA. NEXT JMP, compiles the instruction "Jump to the address of NEXT".

;C makes a couple of error checks, and unsmudges the dictionary header. It compiles nothing.

Next

FORTH interprets your definitions under control of the address interpreter, called NEXT.

At the end of a code definition, control **must** return to NEXT, or to one of a few alternatives which manipulate the stack before returning to NEXT (see Chapter 9 for a reminder of these).

For 6502 systems, NEXT is returned to with a straightforward JMP, instruction.

Security

A reasonable number of checks are carried out on your assembly code, but they are by no means exhaustive.

a) All parameters put on the stack in a CODE definition must be removed before exit.

b) Address modes must be legal for the opcodes.

If an assembly error occurs ; C will not 'unsmudge' the definition, so you will not be in danger of executing a definition containing errors.

Things to beware of are the use of Ø= and Ø< which have different meanings in the Assembler than in FORTH.

6502 Opcodes

All the standard opcodes are supplied, in 4 groups:

- simple opcodes
- multimode opcodes (2 groups)
- branch opcodes

The multimode opcodes require an operand (on the stack), and an address mode.

If no address mode is given, the default is Absolute address. The assembler tries to use Zero Page mode where possible and allowable.

The address mode symbols are as follows:

.A	accumulator	no operand
#	immediate	1 byte
,X	indexed X	Z–page or absolute address
,Y	indexed Y	Z–page or absolute address
X)	indexed indirect X	Z–page address
)Y	indirect indexed Y	Z–page address
()	indirect	absolute indirect

non absolute address

Examples

Here are some examples of assembler statements in FORTH, and in "normal" form. Note that the operand comes first, then an address mode (if any) and finally the mnemonic. All words are separated by spaces, as is normal for all FORTH source text.

		FORTH	NORMAL
	.A	ASL,	ASL A
1	#	LDY	LDY #1
TEMP	,X	STA,	STA TEMP,X
TEMP	,Y	CMP,	CMP TEMP,Y
6	X)	ADC,	ADC (6,X)
TABLE	)Y	LDA,	LDA (TABLE),Y
POINT	()	JMP,	JMP (POINT)

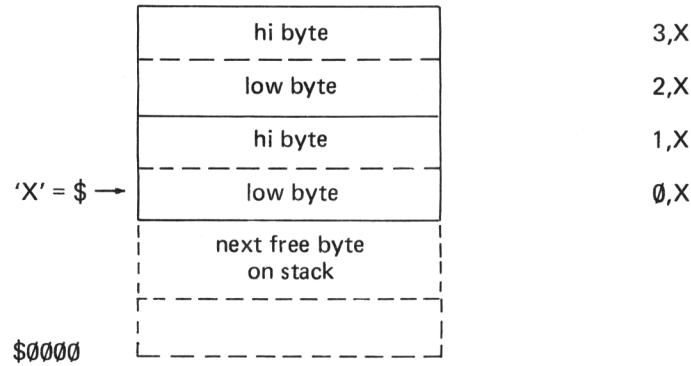
Note: .A for accumulator, to distinguish it from the hexadecimal number A

Accessing the Stacks

The data stack is on Zero Page from locations $9E down to $2Ø. ($ means HEX addresses).

Items on this stack are 16 bit quantities, placed in the normal 6502 way with the low byte at the lower address, followed by the high byte. This allows the use of the (Ø,X) address mode to access memory when the number on the stack is an **address**. X is the stack pointer, which always indicates the current bottom of stack. Decrementing X twice makes room for a new stack item, incrementing twice will 'remove' one. This allows use of n,X mode to access the stack, and (n,X) to use the stack as a pointer.

ZERO PAGE



For example, to add the two bottom stack items together (see also stack diagram):

```
     CLC          ;
Ø ,X LDA          ;
2 ,X ADC          ;    add two LS bytes
2 ,X STA,         ;    & store answer
1 ,X LDA          ;
3 ,X ADC,         ;    add MS bytes
3 ,X STA,         ;    & store answer
POP  JMP          ;    exit losing old top of stack
```

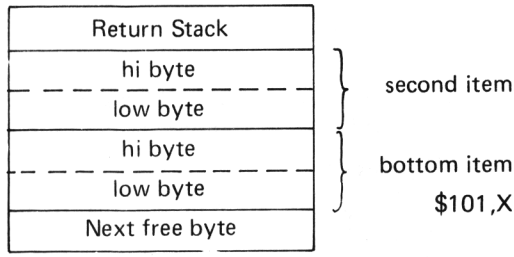
The return stack (in Page 1) grows down from $1FE. The 6502 register (S) points to the **next free** location, and addresses are pushed onto the stack in the same order as above. To access an arbitrary byte on the return stack, the current return stack pointer must be brought into X, so X must be saved first.

```
i.e.: XSAVE  STX,        (save it)
             TSX,        (get SP)
```

at this point; 101 ,X LDA, (101 is in HEX)

will get the most recent item shoved onto the machine return stack.

102 ,X LDA, will get the next byte up etc.



FORTH registers

There are several FORTH 'registers'. For the 6502, these are special zero page locations which are are available only at assembly code level!

They all have names, which when used in the assembler, return the appropriate address.

IP = Interpretive Pointer. This is FORTHs program counter, and points to the next FORTH address to be interpreted by NEXT.

W = Address of the pointer to the code field of the dictionary definition just interpreted by NEXT. W–1 contains $6C – the indirect jump opcode.

JMP W–1 thus performs an indirect jump on a code field to the machine code for the definition.

UP User Pointer. Contains the base address of the User Area.

N A scratchpad area of 9 bytes from N–1 to N+7.

XSAVE Byte buffer in which to save the X register.

CPU Registers

When FORTH executes NEXT and enters a machine code routine, the following conditions apply:

1. The Y index is Ø and may be used freely.
2. The X index points to the low byte of the bottom item on the data stack (relative to $1ØØ).

3. The stack pointer 'S' points to the **next free** byte in the return stack.

4. The accumulator contents are undefined, and ACC may be used freely.

5. The processor is in binary mode with interrupts enabled, and expects to return in that state.

The 'N' Workspace

When extra 'registers' are required, the 'N' area may be used for data storage or as pointers, for example. The assembler word 'N' returns a Zero Page address. Conventionally, N–1 holds a byte count, and N, N+2, N+4, N+6 are four pairs which may hold 16 bit values. Routine 'SETUP' is provided to move values to the 'N' area. It is **important** to note that many FORTH procedures use N, which can only hold values within a single definition, NEVER expect a value to remain in N if you leave one definition and enter another!

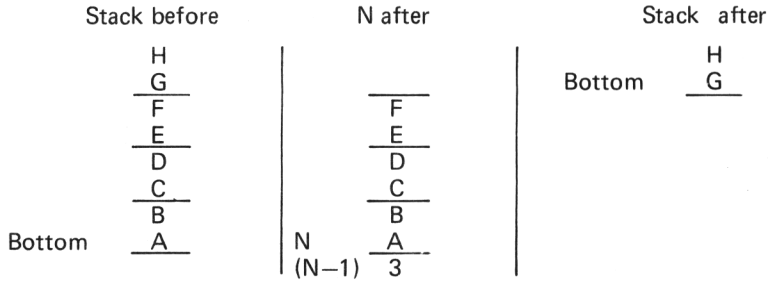
Example:

```
CODE TEST–PORT
6 # LDA, N1 – STA,          (set a counter at N–1)
BEGIN, PORT BIT,            (check an address)
N 1 – DEC,                  (count decrement)
Ø= UNTIL, NEXT JMP, ; C     (loop till count of Zero)
```

'SETUP'

If you need to move stack values to the N area, SETUP is a subroutine provided for this. On entering SETUP, the accumulator contains the number of 16-bit items to be moved, so A can only be 1,2,3 or 4.

e.g. 3 # LDA, SETUP JSR,



Control Flow and Loops

This FORTH assembler allows you to use either branch instructions and labelled statements, or the use of 'pseudo-control' instructions (macro instructions). The latter approach is preferred, as it will result in better structure of your code.

Method 1: Labels may be declared by the pseudo operation LABEL – for example:

LABEL FRED Ø # LDA, FRED BEQ,

Is an infinite loop. The symbol FRED is entered in a **small** symbol table, as a name and the current assembly address. The later use of FRED pushes this address to the stack. If you overflow the symbol table, all hell will break loose!

During debugging, the symbol table may be cleared (wholly or partially) by the KILL xxx instruction, which operates like FORGET xxx within the symbol table.

Method 2: All the FORTH high level control words (except DO..... LOOP) are replicated in the assembler IF, ELSE, ENDIF, BEGIN, UNTIL, AGAIN, WHILE, REPEAT, and are used in the same way.....

EXCEPT for IF, WHILE, UNTIL,. The high level versions test a truth value on top of the stack. The assembler versions test a bit in the 6502 processor status word (PSW). You must specify which 'bit' to test. The available ones are as follows:

CS		test carry set	C = 1
VS		test overflow set	V = 1
Ø<		test negative set	N = 1
Ø =		test zero set	Z = 1
CS	NOT	test carry clear	C = Ø
VS	NOT	test overflow clear	V = Ø
Ø<	NOT	test negative clear	N = Ø
Ø =	NOT	test zero clear	Z = Ø

For example:

PORT LDA, Ø = IF, <a> ENDIF,
reads 'PORT' and executes <a> if equal to zero (Z = 1).

PORT LDA, Ø = NOT IF, <a> ELSE, <b> ENDIF,
reads 'PORT' and executes <a> if non-zero, else <b>.

Similarly for loops:

6 # LDY , BEGIN, PORT DEC, DEY, Ø = UNTIL,
will decrement port until Y reaches Ø.

6 # LDY, BEGIN, DEY, Ø = NOT WHILE, PORT DEC, REPEAT, is similar.

NOTE A ELSE, AGAIN, REPEAT, manufacture an unconditional branch by compiling the code CLV, label BVC, so you can't use these if you are manipulating the processor V bit yourself. The alternative is to rewrite the macro's to use JMP instead.

NOTE B Out of range branch offsets ARE NOT TRAPPED in this version.

ASSEMBLER GLOSSARY

Primitives

$MB Constant for storage of a mode byte for the instruction being assembled.

$MM Constant for storage of addressing mode mask.

$GM			---	nl	returns that part of SMB relating to addressing mode.
$GB			---	nl	returns that part of SMB which indicates Ø,1 or 2 additional bytes to be compiled.
$!M	n	n_2	---		stores n, n_2 into SMB & SMM
$AZ			---		converts SMB from an Absolute to a Zero Page mode byte.
$CA		addr	---	addr	t/f returns 'true' if 'addr' is in Zero Page.

$SD Sets default values (for Absolute address) into SMB and SMM.

$CC	(nl) bytes	final opcode	---		Compiles the opcode and optional 1 or 2 byte arguement into place.

$ME Error exit for an illegal addressing mode.

$PT	(nl)	bare opcode	---		gets the relevant arguments, completes the opcode, and calls SCC.
$CM		address mode	---		checks the current instruction addressing mode for legality, exits via SME if illegal.
$DC			---		converts an Absolute address mode to Zero Page if possible

Definning Words

$$M Defines an addressing mode type.

$IM Defines an implied (no argument) opcode type, and compiles the opcodes.

$BR Defines a 'branch' opcode type, and effects compilation with a computed offset. **(No range check).**

M1	Defines a type 1 opcode/mnemonic and effects the compilation.
M2	Defines a type 2 opcode/mnemonic and effects the compilation.

System Constants

EQU	Shortform redefinition of 'CONSTANT'.	
CS	Ø = Ø < VS	Define branch opcodes as constants.
XSAVE N IP W UP	Are the FORTH registers . addresses.	
NEXT PUSH POP SETUP PUSHØA POPTWO PUT		Exit paths etc.

Pseudo-control Constructs

NOT	br-op ——— br-op	inverts the sense of a branch opcode.
e.g.	VS NOT Ø = NOT	is equivalent to VC is equivalent to Ø ≠
IF		+ condition e.g. Ø = IF, PHA,
*ELSE		
ENDIF,		
BEGIN,		
UNTIL,		+ condition e.g. Ø < UNTIL, ...
*AGAIN,		
WHILE,		+ condition e.g. CS WHILE, ...
*REPEAT,		

These pseudo structures may be used to form loops etc., just as the high level versions do.

* These simulate an unconditional branch with the combination CLV, BVC label.

LABEL	adds a label to the assembler "symbol" table e.g. LABEL FRED PHA, FRED BEQ,
KILL FRED	removes all labels back to and including 'FRED' from the "symbol" table. Message "SYM ERR" if 'FRED' exists but is not in the table.

Additions to the FORTH Vocabulary

CODE	---	Defines the start of an assembly code procedure e.g. CODE TEST1.
;C	---	Ends a CODE procedure or ;CODE

so :CODE ..... ;C
: ..... ;

form a pair like

The assembler also patches the ;CODE procedure in FORTH to point into the assembler.

Glossary Overview

The Glossary of instructions is divided into functional groups, each of which we shall briefly discuss below. Please ensure you are familiar with the first page, showing the layout of each entry, and the symbols for various types of parameter.

1. **Single Precision Arithmetic Operators**

   These should be self-explanatory; note, however that there are various combination operators, and three logical operators (AND, OR, XOR) included here.

2. **Double Precision**

   Working on 32 bit integers.

3. **Mixed Precision**

   i.e. some operands are 16 bit, some 32 bit.

4. **Bases**

   Self-explanatory.

5. **Comparison**

   These operators return a boolean truth flag. Note that while false = Ø, true is defined as **non zero.** Often true = 1, but this is **not a general rule.**

6. **Stack Operators**

   These allow you to move around the top few stack items. Note that R and I have the same effect, but by convention, I is reserved for use within a DO ..... LOOP to indicate that it returns the current loop counter value.

7. **Memory Operators**

   These allow you to manipulate memory locations directly. Note the different operators for byte and word memory access.

8. **Terminal I/O**

   This is possibly the least satisfactory area of FORTH. Just about all the necessary functions are available but they are all done by separate Words.

   Note that 'EXPECT' is a generalised text input word, and ."xxx" is the literal string output operator.

   Character strings in FORTH are stored with a one byte character count at the start (so max length = 255 chars). To print a string knowing its start address, COUNT fetches the 'count' byte to the stack and adjusts the address ready for TYPE.

UPPER is in the dictionary, but has been left 'disconnected'.

9. I/O Formatting

These commands chiefly are used to turn numbers into strings of printing characters, but with much more flexibility (See example in chapter 3).

10. 'Disc' I/O

These should be reasonably clear.

11. Printing

These commands view sections of the 'disc' – see also the PRINT UTILITY.

12. Vocabularies

13 Control Structures

The hi-level constructs available.

14. Defining Words

This is the class of words which allows you to add Words to the dictionary. ' : ' has already been described; CONSTANT, VARIABLE and USER allow the definition of name constants and variables.

CREATE is the primitive which generates a dictionary header with a given name. It can be used directly.

The 'defining words' can be considered as 'compiler instructions' i.e. They tell the FORTH compiler to compile a particular type of Word. One of the great powers of FORTH is that you can add defining words of your own, using combinations of <BUILDS DOES>, { : ;CODE } and { <BUILDS ;CODE }.

15. Dictionary Operators

This group of words carries out operations on the dictionary structure, and on the various fields within a dictionary entry.

16. System Commands

Perform various initialisation routines.

17. System Primitives

Various machine code primitives. You should never need to invoke these directly.

FORTH INSTRUCTION SUMMARY

Arithmetic and integer symbols:

	n =	signed 16 bit integer
	u =	unsigned ditto
	d =	signed 32 bit integer
	ud =	unsigned ditto
Other symbols	c =	ascii character hi 9 bits = Ø
	b =	byte hi 8 bits = Ø
	t/f =	boolean flag: Ø = false
	addr =	address (16 bit)

P means this operation is IMMEDIATE i.e. it will ALWAYS execute, even if FORTH is in compile mode.
E means this operation is available for EXECUTION only.
C means this operation is available for COMPILATION only.

All FORTH operations deal with 16 or 32 bit numbers on the parameter stack, taking input values and returning their results. Byte or ASCII characters are handled as 16 bit numbers with the unused high bits set to zero.

LAYOUT OF GLOSSARY

NAME	INPUTS		STACK	RESULTS	DESCRIPTION
+	nl	n2	----	n3	addition
this is the name of the operation	these are the input values required for the operation. The right-most item is the TOP OF THE STACK.			this is the output value returned to the stack. The right-most item is the TOP OF THE STACK.	
	INPUTS			OUTPUTS	

So this entry is called + It requires two input quantities which are destroyed by the operation. It returns one output quantity – the signed sum of the inputs.

SINGLE PRECISION ARITHMETIC OPERATORS

+	nl	n2	---	n3		n3=n1+n2
–	nl	n2	---	n3		n3=n1–n2
*	n1	n2	---	n3		n3=n1*n2
/	n1	n2	---	n3		n3=n1/n2 truncated
*/	n1 n2	n3	---	n4		n4= n1*n2/n3 31 bit intermediate product
/MOD	n1	n2	---	rem	quot	n1/n2
MOD	n1	n2	---	rem		remainder of n1/n2 i.e. modulo n2 divide

*/MOD	n1	n2	n3	---	n4(r)	n5(q)	as */
MINUS			n1	---	−n1		change sign
MAX		n1	n2	---	n3		n3 is greater of two
MIN		n1	n2	---	n3		n3 is lesser of the two
ABS			n	---	u		leave absolute value
+−		n1	n2	---	n3		apply sign of n2 to n1 leave result as n3
S−>D			n	---	d		sign extend
1+			n1	---	n1+1		increment
2+			n1	---	n1+2		add 2 to integer
AND		n1	n2	---	n3		bitwise and
OR		n1	n2	---	n3		bitwise or
XOR		n1	n2	---	n3		bitwise xor

DOUBLE PRECISION OPERATORS

D+	d1	d2	---	d3		d3=d1+d2
D+−	d1	n	---	d2		as +− on double number
DABS		d	---	ud		as ABS on double number
DMINUS		d	---	d		change sign of double

MIXED PRECISION OPERATORS

U/	ud1	u1	---	u2(r)	u3(q)	unsigned divide
U*	u1	u2	---	ud3		unsigned multiply
M/	d1	n1	---	n2(r)	n3(q)	signed divide
M*	n1	n2	---	d1		signed multiply
M/MOD	ud1	u2	---	u3(r)	ud4(q)	unsigned divide

NUMBER BASES

HEX	set number base to 16
DECIMAL	set number base to 1Ø

COMPARISION OPERATORS

Ø<		n	---	t/f		leaves true if n negative
Ø=		n	---	t/f		true if n zero (reverses truth value)
>	n1	n2	---	t/f		true if n1>n2
<	n1	n2	---	t/f		true if n1<n2
U<	u1	u2	---	t/f		unsigned true if u1<u2
=	n1	n2	---	t/f		true if n1=n2

STACK OPERATORS

ROT	n1	n2	n3	---	n2	n3	n1	bring third to top
DUP			n1	---	n1	n1		duplicate top of stack
SWAP		n1	n2	---	n2	n1		swap top two items
DROP			n1	---				remove top item
OVER		n1	n2	---	n1	n2	n1	duplicates second onto top
PICK			n1	---	n2			fetches n2, which is the n1'th entry down on the stack

–DUP	n1	---	n1		if n1–Ø
	n1	---	n1	n1	duplicate if non zero
R		---	n		copy top of return stack to parameter stack
I		---C	n		same as R
>R	n	---C			Puts top of stack onto return stack.
R>		---	n		removes top of return stack to parameter stack
SP@		---	addr		leaves address of top of stack (before this item was added)

!CSP (compiler use only) saves stack position in variable CSP for compile time checks

MEMORY OPERATORS

CMOVE	from	to	count	---	move count bytes; starting at 'from' upwards
?		addr		---	print on terminal contents of 'addr' as signed integer
BLANKS	addr	count		---	fill memory with 'count' spaces starting at 'addr'
ERASE	addr	count		---	fill memory with 'count' zero's
FILL	addr	count	b	---	fill memory from'addr' with 'count' bytes 'b'
C!	b	addr		---	store byte at addr
!	n	addr		---	store word at addr
C@		addr		--- b	fetch byte from addr
@		addr		--- n	fetch word from addr
+!	n	addr		---	add n to location addr

TERMINAL I/O

.		n	---	print n as signed single integer with a trailing blank
.R	n1	n2	---	print n1 right justified in field width n2
D.		d	---	print signed double integer
D.R	d	n	---	print d right justified in field width n
SPACES		n	---	print n spaces on terminal
WORD		c	---	read input stream until the first non-delimiter character (c is the selected delimiter) is found. Transfer the character string that follows to HERE until the next delimiter is located. The first byte of the packed string at HERE is the length byte.

QUERY		---		input 8Ø chars (or until CR) to TIB. At exit from QUERY, IN = Ø
EXPECT	addr count	---		read chars from terminal storing at 'addr' upwards until CR received, or 'count' expired. A Null character (Ø) is added to the end of the string
."		---P		compiles in line string terminated by " executes immediately outside a definition.
COUNT	addr1	---	addr2 n	for a string at addr1 with first byte set to string length: returns length n and start. Use before a TYPE to get length byte out.
SPACE				print one space
CR				print CR
?TERMINAL		---	t/f	tests for 'break' (ctrl C) from terminal leaves true if break received, else false. ?TERMINAL does not wait for a key, it inspects the input buffer to see what the last char was.
KEY		---	c	returns next char received from terminal. KEY waits for a key to be pressed.
EMIT	c	---		sends c to terminal.
TYPE	addr count	---		transmit a string of count chars starting at addr upwards to the terminal device.
MESSAGE	n	---		print error message n to terminal if variable 'WARNING' = Ø will print 'n' only.
.LINE	line scr	---		print on terminal; line no 'line' of screen 'ser'.
(		---P		accept comment terminated by) space after (required
ID.	addr	---		print definition name from name field address on the stack
DIGIT	c n1	---	n2 true	convert char c to binary n2 using current BASE n1
	c n1	---	false	if conversion invalid

I/O FORMATTING

#S	d1	---	d2	convert d1 to ascii in output buffer, by repeated calls to #, until d2 is zero. Use between <# and #>.
#	d1	---	d2	Conversion proceeds from least significant end of d1. From d1, generate the next ascii char to output string. d2 is quotient after division by 'BASE'.
SIGN	n d	---	d i	if n negative, stores ascii minus sign in output buffer before a numeric output string. Use between <# and #>.
<# start numeric output conversion e.g. <# #S SIGN #>.				resulting string stored at PAD downwards.
#>	d	---	addr count	terminate numeric conversion leaving suitable parameter for 'TYPE'.
NUMBER	addr	---	d	convert ascii string at addr, to a signed double number using the current 'BASE'. The first byte of the string must be the length byte. If a decimal point is encountered, its position is returned in DPL. If DPL = –1, no decimal point was found. This allows you to identify 32 bit input by putting a '.' in the number.
HOLD	c	---		insert ascii char into numeric output string. Use between <# and #> only.
PAD		---	addr	returns start address of temporary text buffer.
–TRAILING	addr1 n1	---	addr2 n2	adjusts count n1 of string to suppress any trailing spaces in the string.

DISC I/O

LOAD	n	---		begin interpretation of screen n. Loading will terminate at end of screen or at {;S}.
-->		---P		continue interpreting with next screen.
BLOCK	n	---	addr	leaves memory address of the buffer containing disc block n. The block will be fetched from disc if not already resident. BLOCK incorporates the OFFSET adjustment.
BUFFER	n	---	addr	get next buffer to use and assign to block n. If marked as updated, contents are first rewritten to disc. addr is first data cell. BUFFER does NOT incorporate the OFFSET adjustment.

DRØ
DR1
DR2
DR3 installation dependent commands to preset variable OFFSET to allow for drive selection.

EMPTY–BUFFERS mark all buffers as empty. Updated buffers are not written to disc. Used as initialisation procedure.

FLUSH force all updated buffers to be written to disc.

UPDATE mark the most recently used disc buffer (pointed to by PREV) as updated.

+BUF	addr	---	addr2	t/f	advance to next disc buffer addr2. t/f is false if addr2 points to same buffer as PREV
.LINE	line scr	---			see under TERMINAL I/O disc primitive to transfer block number 'blk' at address 'addr' to (flg=Ø) or from (=1) disc NOTE 'blk' is a FORTH 1kbyte block.
PREV		---	addr		system variable contains address of disc buffer most recently referenced.

USE		---	addr	system variable contains addr of next buffer to be used (the one least recently accessed).
B/SCR		---	n	constant which contains the number of disc buffers per 1Ø24 byte screen.
B/BUF		---	n	constant which contains the number of bytes per buffer (also disc sector size)
LIMIT		---	n	constant address of top-of -disc-buffers +1.
FIRST		---	n	constant leaves address of first disc buffer.

PRINTING

TRIAD	n	---		displays the three screens including screen n starting with a screen evenly divisible by 3.
LIST	n	---		display screen n. variable SCR will hold n.
INDEX	from to	---		print the first line of each screen in the given range. Use to view the comment lines.

VOCABULARIES

VLIST	list all definition names in CONTEXT vocabulary. (CONTEXT vocabulary is the one which is searched first i.e. it is the EXECUTION vocabulary).			
DEFINITIONS	used in form cccc DEFINITIONS to set CURRENT equal to CONTEXT. Executing cccc made it CONTEXT and DEFINITIONS forces CURRENT to equal CONTEXT. (CURRENT is the vocabulary to which new Words are added).			
FORTH		---P		sets CONTEXT vocabulary to be FORTH.
VOCABULARY	used in form VOCABULARY cccc to create a vocabulary definition cccc. Subsequent use of cccc makes it the CONTEXT vocabulary. VOCABULARY definitions should be declared IMMEDIATE.			
CURRENT		---	addr	user variable leaves addr of pointer to first item in current vocabulary.
CONTEXT		---	addr	user variable leaves addr of pointer to top of context vocabulary (searched first)

CONTROL STRUCTURES

IF		t/f	---PC	run time
ELSE			---PC	
ENDIF			---PC	

the above form the conditional execution structures:
IF (true part) .....ENDIF
IF (true part) .....ELSE (false part) .....ENDIF
selection of true or false part is made from top-of-stack boolean as shown for IF (run time).

DO	n1	n2	---PC	run time
LOOP			---PC	run time note that the DO.....LOOP parameters are stored on the return stack during execution of the LOOP.
+LOOP	n1		---PC	run time

The above three form a looping construct which tests for exit/continue at the bottom of the loop:
DO.....LOOP at run time 'DO' need variables n1 (loop limit) and n2 (initial index value).

LOOP increments the index by 1 and if the result is < the limit n1 the loop is re-entered at the top.

DO.....n1 +LOOP as above except the top-of-stack signed value n1 is added to the loop index. n1 must be on the stack each time through the loop. The branch back to the DO takes place until the new index is >= the limit (if n1 >= Ø) or until the new index is <= the limit for n1 <Ø.

LEAVE	---C	force termination of a DO.....LOOP by setting the loop limit equal to the index.
I	---C n	used within a DO.....LOOP to copy the loop index to the stack (see R also).

The next 6 instructions provide the more general form of looping construct.

BEGIN		---P	marks the start of a loop.
WHILE	t/f	---PC	conditional exit point. false forces loop exit.
REPEAT		---PC	loop terminator after WHILE
UNTIL	t/f	---PC	loop teminator which exits if true, else loops.

AGAIN		---PC	loop terminator unconditionally loops to BEGIN
BACK	addr	---	compiler only. Compiles a backward branch offset.

The following loops may be constructed:

BEGIN.....UNTIL	loop is executed until a true flag is top of stack at UNTIL i.e. lóop UNTIL true
BEGIN.....WHILE.....REPEAT	boolean is tested by WHILE and exits from loop if false. i.e. WHILE true, continue loop.
BEGIN.....AGAIN	uncondition loop

DEFINING WORDS

:	---PE	begin colon definition of new procedure creating a new dictionary header and setting 'compile' mode.
;	---PC	terminate colon definition as a high level 'word' and terminate 'compile' mode. Compiles the run-time ;S into the dictionary.
;CODE	---PC	terminate colon definition as defining word with machine code execution code following.

the sequence : FRED.....; compiles a dictionary entry named FRED whose CFA points to the routine which executes a sub-routine-like function and leaves the address interpreter pointing to the first 'word' in 'FRED's PF

the sequence : JIM. . ;CODE code compiles a new defining word JIM. When 'JIM' is executed in the form JIM JACK a dictionary entry JACK is created whose CFA points to the machine language routine 'code' defined IN 'JIM'.

N.B. ;CODE REQUIRES AN ASSEMBLER TO COMPILE THE CODE PART *****

CONSTANT n ---

Use in form n CONSTANT MARY to compile a dictionary entry 'MARY' whose PFA contains the constant value n. The CFA points to a routine to push the value 'n' onto the stack when 'MARY' is executed.

VARIABLE n ---E

Use in the form n VARIABLE MICK to compile a dictionary entry 'MICK' whose PFA contains the variable n. The CFA points to the routine to push the PFA onto the stack when 'MICK' is executed.

USER n ---

Use to create an entry in the user area. n USER FLOB generates a dictionary entry 'FLOB'. Its PFA contains the value n which is the offset into the user area for this entry. When 'FLOB' executed it pushes the actual memory address of the entry onto the stack.

VOCABULARY ---E

USE as VOCABULARY GROT IMMEDIATE to create vocabulary definition 'GROT'.

<BUILDS.....DOES> ---C

Use within a colon definition : SPIT <BUILDS.....DOES>..; to generate a new defining word 'SPIT' which 'builds' dictionary entries according to the <BUILDS part and whose execution procedure is the high level instructions which follow DOES> up to the semicolon. This is best explained by example and trial and error!

Note that when the new Word executes, DOES> leaves the Parameter field address of the new Word on the stack.

CREATE ---

This is the primitive which 'creates' a dictionary header at 'here'. The header is 'smudged' and the CFA points to the PFA (i.e. to the address following the CFA). CREATE may be used directly CREATE CC to generate the dictionary header 'CC'. Use with caution.

DICTIONARY OPERATORS AND COMPILATION DIRECTIVES

COMPILE when the dictionary entry containing 'COMPILE' executes, the CFA of the word following it is compiled into the next dictionary location.

[COMPILE] use in a definition e.g. : xxx [COMPILE] yyyy ..; forces yyyy to be compiled when normally immediate

[immediate. Suspends compilation state within a definition to allow a computation to take place.

] Resumes compilation state e.g. : xxxx [some Words] more-words ;

The Words between [] are executed at compile tiem. This allows compile time setting of parameters.

IMMEDIATE mark the most recent dictionary definition as immediate. i.e. the precedence bit is set in the header

TOGGLE	addr b	---		compliment contents of 'addr' by bit pattern 'b' TOGGLE is used by SMUDGE

SMUDGE used during a definition to toggle the smudge bit in the definition name field used automatically by ' : ' and ' ; ' etc. A smudged Header cannot be found in a dictionary search.

LATEST		---	addr	Leave the PFA of the top-most CURRENT word
PFA	nfa	---	pfa	convert the nfa to the pfa of the word
LFA	pfa	---	lfa	convert the given pfa to the lfa
CFA	pfa	---	cfa	convert the given pfa to the cfa
NFA	pfa	---	nfa	convert the pfa to the nfa
'		---P	addr	used in form ' FRED to leave pfa of FRED. If compiling, the pfa is compiled as a literal
TRAVERSE	addr1 n	---	addr2	traverse across a name field header n=direction. (1 = low to high address, −1 = high to low addrs
ID.	nfa	---		print the definitions name from the nfa
−FIND		---	pfa b true	if found, b=byte count of name
		---	false	if not found

−FIND accepts next text word in input stream (delimited by blanks) and tries to find a match in the dictionary.

ALLOT	n	---		reserve dictionary space for n bytes by adding n to the dictionary pointer
,	n	---		store n in the next available dictionary cell and increment the dictionary pointer. (comma)
C,	b	---		as for { ,} except does single byte
LITERAL	n	---PC		compile n as 16 bit literal value
DLITERAL	d	---P		compile a 32 bit literal

At execution time, LITERAL an DLITERAL will push the stored value onto the stack.

HERE ––– addr returns address of next available dictionary cell pointed to by variable DP

;S Stop interpretation of a disc screen, {;S} is also the run time routine compiled by {;}

→ Continue interpreting on next disc screen

SYSTEM COMMANDS

MON Exit to BASIC. Use with care!

FORGET Use in form FORGET FRED . Deletes definition FRED and all following definitions from dictionary

+ORIGIN n ––– addr for offset n into origin memory area leave actual memory address

RP! Initialise return stack pointer

SP! Initialise parameter stack pointer

QUIT clear return stack, stop compiling and return to terminal input state

ABORT clear both stacks, return to terminal control printing start up greeting message

COLD Cold start procedure, initialises stacks, user variables dictionary pointer and restarts via ABORT

EXECUTE addr ––– execute definition whose CFA is on stack

INTERPRET the outer text interpreter which compiles or executes the input stream (terminal or disc depending on STATE)

SYSTEM PRIMITIVES

(LINE) n1 n2 ––– addr count convert line n1 of screen n2 to appropriate buffer address and char count, (64 max)

(ABORT) actual abort procedure. Will also execute after an ERROR if WARNING is –1.

(NUMBER) d1 addr1 ––– d2 addr2 convert ascii string at addr1+1 to double number d1, finally left as d2. Addr2 is first unconvertible character

(.") The run time procedure compiled by ."

(FIND) addr1 addr2 ––– pfa b true if found
addr1 addr2 ––– false not found
(FIND) searches the dictionary starting with a NFA at addr2 matching to text at addr1

(DO) Run time procedure compiled by DO

(LOOP) ditto for LOOP. The next word compiled is the branch offset

(+LOOP)	ditto for +LOOP
ØBRANCH	run time procedure for conditional branch, with following branch offset
BRANCH	ditto for unconditional branch
CLIT	run time procedure to push a character literal to the stack
LIT	ditto for word literal
ENCLOSE	addr1 c --- addr1 n1 n2 n3 text scanning primitive used by WORD. Enclose reads the input stream from addr1 upwards, looking for delimiter 'c'. Initial occurrences of 'c' are ignored. When a non-delimiter is found, it then searches for the next occurrence of 'c' after the intervening text string, and then exits leaving offset n1 from addr1 to the first non-delimiter offset n2 from addr 1 to the first delimiter after the string. offset n3 from addr1 to the first char not included in the search. Note Ascii NULL (Ø) is treated as an unconditional delimiter if encountered in the search.
(;CODE)	run time procedure compiled by {;CODE}

ERROR PROCEDURES

ERROR	n --- in blk	execute error notification of error n and restart system leaving in and blk.
	If WARNING = Ø, numbered error messages are produced. If WARNING = 1, message text is taken from screens 4 and 5 of drive Ø. If WARNING = –1, the system ABORTS.	
?ERROR	t/f n ---	issue error n if flag true.
?STACK	issue error message if stack out of bounds	
?EXEC	ditto if not executing	
?COMP	ditto if not compiling	
?PAIRS	n1 n2 ---	issue error if n1 not equal to n2 is used to indicate paired conditionals
?CSP	issue error if stack position is not same as that stored in CSP	
?LOADING	issue error if not loading	

NAMED CONSTANTS

B/SCR	returns number of disc sectors (blocks) per edit screen (1k)
B/BUF	returns bytes per disc sector
LIMIT	returns memory addr+1 above disc buffers
FIRST	returns memory addr of first disc buffer
C/L	number of characters per line (64 decimal)
BL	returns ascii for 'space'
3	three
2	two

1	one
Ø	zero these are used such a lot they are constants

NAMED VARIABLES

USE	holds address of next disc buffer to be used
PREV	holds address of most recently referenced disc buffer

USER VARIABLES

TIB	holds address of terminal input buffer
WIDTH	contains max width currently allowed in name field. (max is 31 decimal)
WARNING	controls disc messages
FENCE	address below which FORGETting is trapped
DP	contains address of next cell in dictionary
VOC-LINK	contains address of a field in most recent vocabulary structure.
BLK	contains current block number. Ø means take input from TIB
IN	offset pointer into input text. Used mainly by WORD
OUT	count of chars output to terminal. Use for formatting. Is reset to Ø when CR or FF is output to terminal.
SCR	contains screen number last used by LIST.
OFFSET	points to different disc drives as a block offset.
CONTEXT	pointer to top of context vocabulary.
CURRENT	pointer to vocabulary into which new definitions will go.
STATE	compilation state. non zero = >compiling.
BASE	current number base
DPL	after input number conversion, contains number of digits to right of a decimal point if no decimal point is input, DPL = –1
FLD	for control of ouput number field width. Not implemented in fig–FORTH
CSP	used by compiler to store stack position
R#	for use by editor as cursor position offset
HLD	holds address of latest character during output number conversion

EDITOR COMMANDS

SELECTING A SCREEN FOR EDITING

EDITOR n LIST <CR>	lists screen n and selects it for editing
EDITOR n CLEAR <CR>	clears screen n and selects it

TEXT INPUT COMMANDS

P	n	---	Puts following text on line n e.g. 3 P THIS TEXT ON LINE 3<CR>

NEW	n	---	selects and displays line n for insertion. CR terminates and selects next line. Existing lines overwritten.
UNDER	n	---	selects line n+1 for insertion of text original lines n+1 onwards are moved down. Line 15 is lost.

EDIT CURSOR CONTROL

TOP	places edit cursor at beginning of text screen		
M	n	---	Move edit cursor by signed displacement n New cursor line is displayed

LINE MANIPULATION

H	n	---	hold line n in buffer PAD
D	n	---	Delete line n but save in PAD lines n+1 to 15 move up, 15 blanked
T	n	---	Type line n and save in PAD
R	n	---	replace line n with the text in PAD
I	n	---	Insert text from PAD to line n, moving old lines n to 14 downwards.
E	n	---	Erase line n to spaces
S	n	---	Lines n to 14 move down, leaving n blank.

STRING COMMANDS

F text <CR>	search forward from cursor position for string 'text'. The cursor is left at the end of the string and the cursor line displayed. 'NOT FOUND' error given if string not located.
B	move the cursor back by the length of the text string used for Finding. Use to position cursor at beginning of text string
N	Use after F to get to Next occurrence of the same string.
X text <CR>	Find and delete the string 'text'
C text <CR>	Copy the string 'text' into current line at the cursor position
TILL text <CR>	Delete forward from the cursor position to the end of string 'text'
NOTE:	Typing C with NO TEXT will copy a null into the text which will cause compilation to abort later on.

SCREEN MANIPULATION

LIST		see FORTH glossary		
CLEAR		n	---	clears screen n to spaces
COPY	n1	n2	---	copy screen n1 TO n2

L					relists current screen with cursor line
FLUSH					Use at the end of an edit session to ensure all buffers are written out to 'disc'.

EDITOR PRIMITIVES

NOTE: THE USE OF 'CURSOR' REFERS TO THE EDITOR CURSOR NOT THE SCREEN CURSOR.

TEXT		c	---			Accept the following text to PAD 'c' is the text delimiter
LINE		n	---	addr		convert line No. n of current edit screen to address of buffer containing that line
WHERE	n1	n2	---			n2=block number, n1=offset If an error occurs while loading from disc. ERROR leaves n1, n2 to locate the fault. WHERE prints a 'picture' of the error position.
#LOCATE			---	n1	n2.	convert cursor position to line no. n2 and offset on that line n1
#LEAD			---	addr	n	leave addr of cursor line and offset-to-cursor n
#LAG			---	addr	n	leave cursor address and count to end of line
–MOVE	addr	n	---			move a line of text from addr to line n of current screen
–TEXT	addr1	addr2	n2 ---	t/f		compare two strings, length of shorter=n string start addresses are addr1 and addr2 flag is true if they match, else false
MATCH	addr1	n1	addr2	n2 ---	t/f n3	addr1=cursor address, n1= count to end-of-line addr2= string address, n2=string count

MATCH searches forward from addr1 for a string match to the string at addr2 length n2.
It returns true if one found, and n3=change in cursor position
It reurns false if not found, n3=count to end-of-line.

1LINE	---	t/f	Scans current edit line for a match to the text in PAD. flag returned, cursor updated
FIND	---		Search forward from cursor for match to the text in PAD. If no match, issue error and put cursor at top of text screen.
.BS	---		emit a backspace char.
NULL?			
ENTER			
ENTER?			
2DROP	drop a double number		
2DUP	DUP a double number		
2SWAP	SWAP two double numbers		

Optional Words on Cassette Screens

Option Screen 1: Screen I/O

PTC	y x	---	Puts VDU cursor on row Y, character position X of screen.
GTC		--- y x	Returns current cursor position. Puts cursor at start of line.
CLINE	y	---	Clears line y of screen.
IN#		--- n	Inputs a number from keyboard.
PON		---	Route all output to printer.
POFF		---	Switch off printer output.
PICK	n	---	get n'th value on stack and duplicate on top of stack.

Option Screen:Screen 2

Random Numbers

VRND variable containing last random integer
PRND primitive to perform necessary arithmetic

RND		--- n	random integer n returned NOTE n is always positive
SRND	n2	--- n	returns random integer n scaled to lie within the range Ø to n2.
RANDOMISE	n	---	seeds the generator with n

Option Screen 3

1–D ARRAYS

1ARRAY	Builds a one dimensional array e.g. 3 1ARRAY FRED builds an array of 3 integer cells (i.e. offset range Øto 2) The dimension is also stored Executing 2 FRED then returns the ADDRESS of element 2
1CARRAY	As above but with byte cells

Also included for use when debugging is an alternative version of 1ARRAY which range checks the index on execution. An error message is then given if the index is out of range.

Option Screen 4

Case Statement – Numeric Key

This screen contains five Words which allow a 'Pascal' like CASE statement to be compiled. A CASE statement is similar to a multiway branch allowing 1 to N possible choices depending on the value of the input KEY, which for this version is an integer value.

Also included is the option to execute default code if the KEY is not matched in the CASE body.

The 5 Words on this screen are:

(OF)	machine code primitive compiled by OF
CASE	the opening Word of a CASE statement
OF	tests input KEY value for equality with given value
ENDOF	denotes the end of an OF statement
ENDCASE	denotes the end of a CASE statement

examples for use:

```
:FRED        CASE (start a definition FRED and a CASE statement)

             23 OF.........................ENDOF
             12 OF.........................ENDOF
             134 OF........................ENDOF
                DEFAULT CODE
                ENDCASE
```

The CASE statement expects an argument on the stack which is matched in turn to each of the numbers given. If the match is obtained the argument is dropped, and the code between the following OF.....ENDOF is executed, followed by a skip to END CASE which exits. If no match is obtained, the 'DEFAULT CODE' is executed, this being optional.

NOTE that ENDCASE always executes a DROP as its first instruction. This must be allowed for if you put anything into the DEFAULT CODE slot.

This CASE features compile time checks, and can be nested, i.e. further CASE statements can be placed between OF and ENDOF for example.
Stack organisation:

At entry to CASE N --- n is the integer KEY
After successful match --- after executing OF
If no match obtained --- n is dropped at the start of ENDCASE
So the input KEY value is always lost, so DUP it first if you want it later on!

Option Screen 5

Sound Commands

PING	---	
SHOOT	---	Preset sounds. Function as for Basic.
EXPLODE	---	
ZAP	---	

SOUND c p v	---	As for Basic:	c=Channel p=Period
MUSIC c o n v	---	As for Basic:	v-Volume o=Octave
PLAY t n m p	---	As for Basic:	n=Note t=Tone channel(s) n=Noise channel(s) m=Envelope Mode p=Envelope Period

Option Screen 6

High Resolution Graphics

Machine code primitives. Not used directly.

(CURSET)
(CURMON)
(DRAW)
(CIRCLE)
(PATTN)
(CHAR)
(POINT)
(FILL)

Option Screen 7

High Resolution Graphics

CURSET x y fb	---	As for Basic:	x=x position

				y=y position
				fb=Foreground/Background
CURMOV	xr yr fb	---	As for Basic	xr=x relative
				yr=y relative
DRAW	xr yr fb	---	As for Basic	
CIRCLE	r fb	---	As for Basic	r=Radius
PATTERN	n	---	As for Basic	
CHAR	x s fb	---	As for Basic	x=ASCII value
				s=Character Set
POINT	xr yr	---	As for Basic	l=logical value on/off
FILL	b a n	---	As for Basic	b=Rows
				a=character cells
				n=value

Note that for out-of-range errors commands will return without executing. To see if an out-of-range error has occurred, read location $2EØ (Hex) Ø=OK

1=Range Error

TANSOFT LTD.,
3 Club Mews,
Ely,
Cambs CB7 4UN

ORIC PRODUCTS INTERNATIONAL LTD.,
Coworth Park Mansion, Coworth Park,
London Road, Sunninghill,
Ascot, Berks, SL5 7SE

Printed by: S-Print 24 Milton Road, Cambridge. Tel: 311334