ORIC-1

FORTH PROGRAMMING MANUAL

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 commard '.' (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.

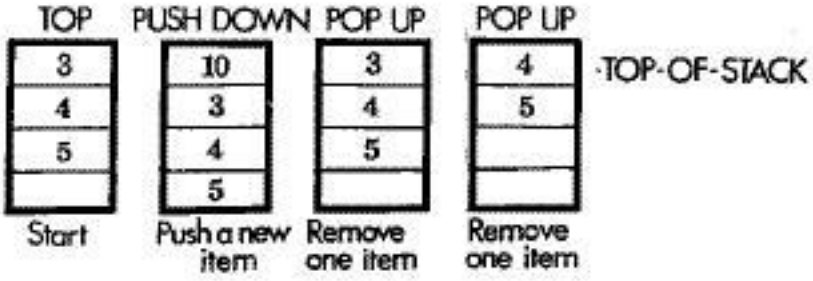


Fig. 1. A push-down stack

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

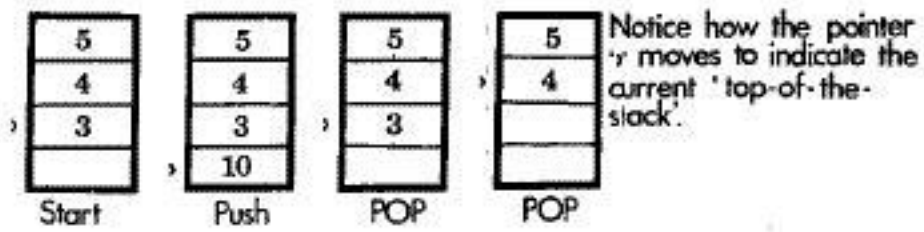


Fig. 2.

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:

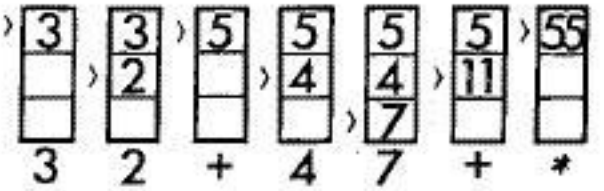
Reading from left to right, FORTH first encounters the 3. This, like all explicit numbers, is pushed onto the stack. Likewise the next item, the 2. The next thing is + which FORTH interprets as follows:

'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 250 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 0 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 (0 to 1023) 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 '0 <' 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 0<                  (copy the number, test its sign)
IF MINUS ENDIF          (change sign if negative)
;                       (end of this word)
```

and then try it

```
10 ABS -VALUE . 10 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

```
: 10CUBES               (name of this word)
0                       (initial count value)
BEGIN                   (start of loop)
CR DUP . DUP CUBE .     (print a number and its cube)
1 +                     (increment the index)
DUP 10 =                (test for index = 10)
UNTIL                   (end of loop, exit if true)
CR DROP                 (throw away final index)
; OK                    (end of word)

10CUBES                 now execute it

0       0
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 30 – 40 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 0 to 15, each of which can hold 64 text characters.

To enter some new text on line 0, enter the command
0 NEW <CR> which 'opens' line 0 for text entry. Anything you type up to the next <CR> will be entered into that line.

By convention, line 0 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
n1 n2 COPY <CR> copies screen n1 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.

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 0 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

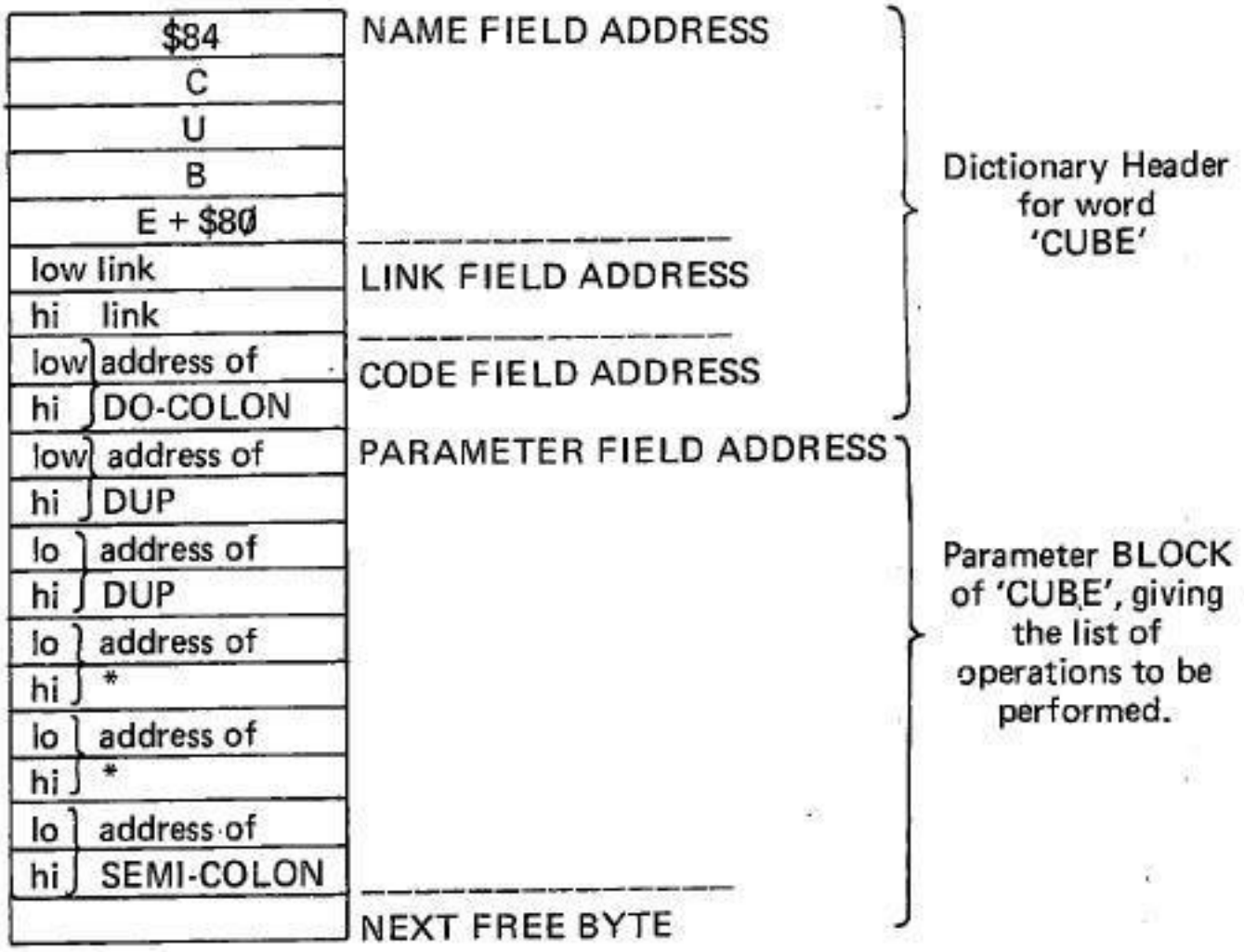
Enters with the starting screen 'from' and 'count' left. Since a PAGE is 3 screens, it generates the updated 'from' and 'count', and the requisite arguments for PRINT–IT, in the form 'from+3' and 'from' i.e. these are 'end' and 'start' screen numbers for a PAGE of 3 screens.

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.



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.

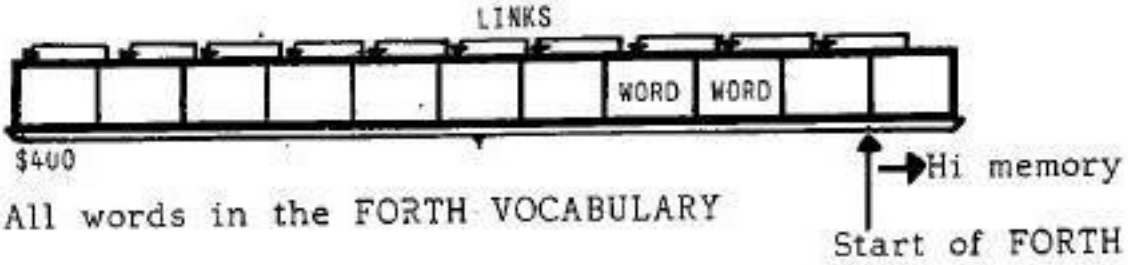
Vocabularies have two main effects: First they increase compilation speed, by allowing dictionary searches to start in the right 'area'. Secondly, you can have Words with the same Name in separate

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

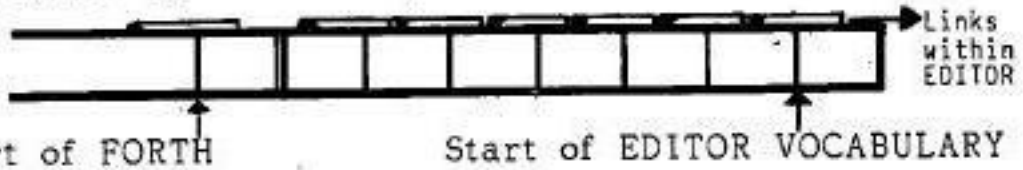
When a new Vocabulary is set up, the Link address chaining is modified to ensure that new Words are compiled into the CURRENT VOCABULARY.

An Example

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

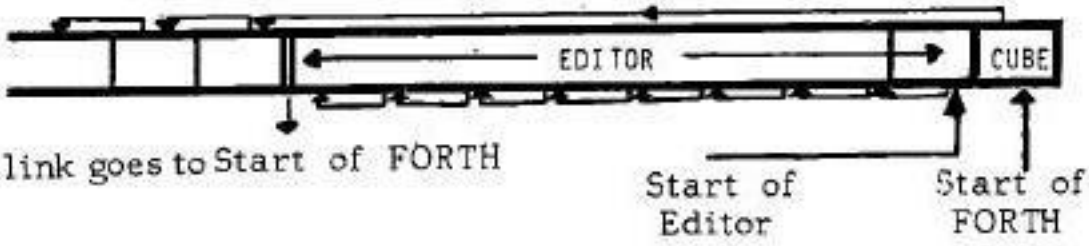


2. On adding the EDITOR, a second Vocabulary exists:



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.

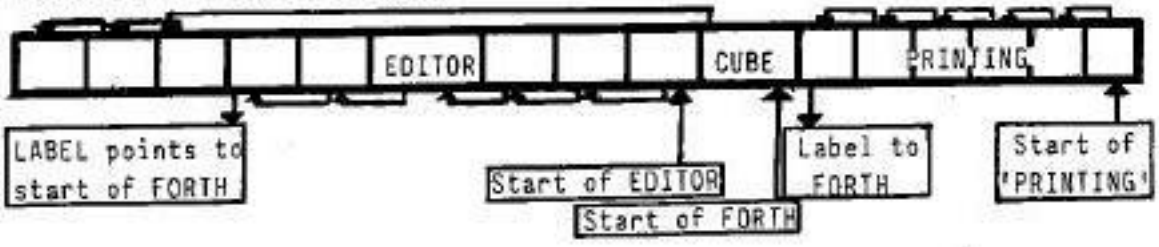
3. Now suppose we add : CUBE to the FORTH vocabulary.



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.

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.

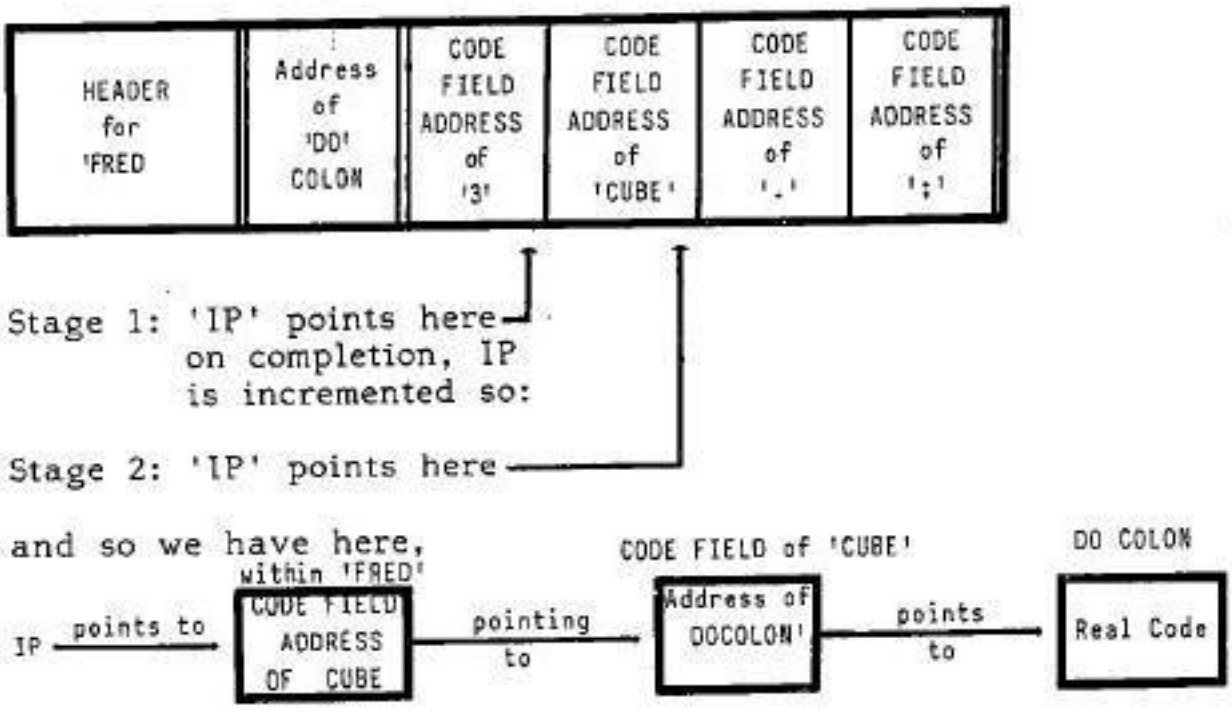
The contents of the CODE FIELD are the address of a real machine language routine to be executed when the word is first called, i.e. the CODE FIELD ADDRESS represents the start of the word.

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, ir 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.

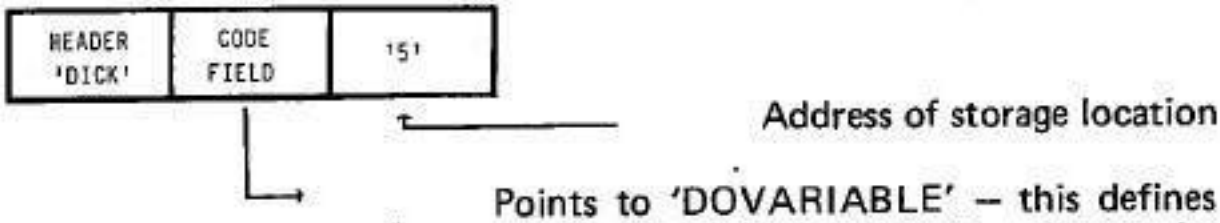
Example 1

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.



the operation of VARIABLE, which is that when DICK executes, the ADDRESS of the storage location is pushed onto the stack.

Example 3

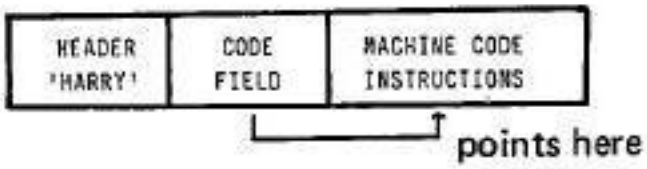
26 USER TOM defines a USER variable, named TOM. User variables are stored in a special block in high memory, and this would define TOM as being the 26th entry in the user block, and when TOM executes, the ADDRESS of the storage location is pushed onto the stack.

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:



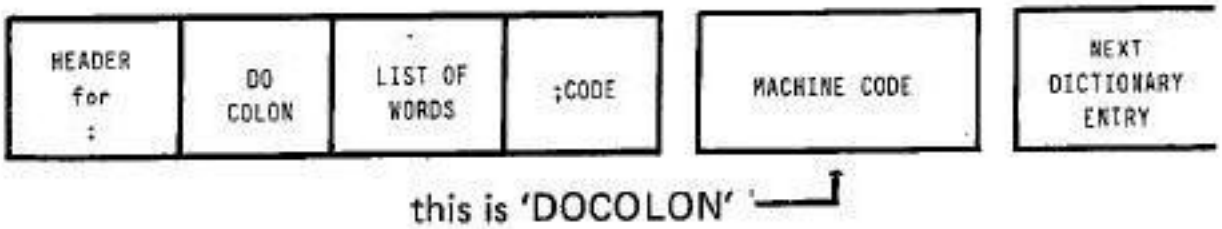
(See also the next chapter)

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 9

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
```