nascom

fig-FORTH f· · the NASCOM SERIES

nas-FORTH 1.1

By
DAVID HUSBAND
2 GORLESTON ROAD
BRANKSOME, POOLE
BH12 1NW
ENGLAND

DEFINTION OF +- WRONG

SHOULD BE

: +- 0< IF MINUS THEN ;

CORRECT BY

HEX 14E6 218A ! DECIMAL

0.0 nas-FORTH Index

TABLE OF CONTENTS

1.0 nas-FORTH

This Manual is not intended to be a tutorial on the use of FORTH, it is intended to tell you things particular to nas-FORTH and about the various improvements and enhancements that have been incorporated in this version for the NASCOM range of computers.

Specific topics included in this section are :

1.1 Nascom Features

The best feature of the Nascom (in my opinion) is the superb screen editing facilities provided at a time when such things could not be taken for granted. It is fundamental to the Nascom screen editing routines that there is a routine that, when invoked, returns the address of the first character of the line where the cursor is currently positioned. All screen editing functions are based on the use of this routine (CPOS). This technique works well on the Nascom because the screen is memory-mapped, and consequently very easy to manipulate. FORTH, however, is fundamentally different in the way that I/O is handled. It uses the old 'serial', one character down a tube type of I/O, and one would be justified in asking why this method is used. The reason is that there are very powerful advantages in using the old method. Few people realise that FORTH is actually a multi-user, multi-tasking language, and that the FIG version has been modified for microcomputer users. FORTH's use of the I/O is actually an advantage if you want to create more sophisticated applications (like support a remote terminal).

Because of FORTH's I/O handling, the four cursor control codes of the Nascom have been trapped and therefore do not work. This relates to the character input routines. On the character output routine EMIT, the eighth bit is not stripped off as it usually is, so that the Nascom Graphics ROM can be used. An extra word, PAUSE, has been added to some of the screen output routines. The effect of this is to help you when you are listing something to the screen ; VLIST for example. What PAUSE allows you to do is to hit any key while output is going to the screen, and the output will pause, and then hit any key to start output again or ESC to terminate the output. This is a great convenience for some types of output.

1.2 Stack Display

A very useful utility provided is the automatic stack display. This only operates when there is something on the stack and it then shows you what is there. This is most useful when you are first learning FORTH as you can play about with various words, such as DUP, ROT, SWAP, etc. and discover the effect on the stack contents. You must also remember that FORTH works in any number base, so you can change from bases by executing words like DECIMAL, HEX, and BINARY. Other words can be defined at will. (See Section 1.8 NUMBER BASES).

Another use for the stack display is in the debugging process. It is very important that items are not left on the stack by mistake, as this will overflow the stack eventually and result in an error message. It is equally important to guard against underflowing the stack as this results in an immediate error message. (See Section 1.9 ERROR MESSAGES).

Example of the Stack Display: <cr> means pressing RETURN
User input in **bold**

```
12   <cr>

Stack Base > 12
ok

34   <cr>

Stack Base > 12 34
ok

+    <cr>

Stack Base > 46
ok
```

1.3 Tape Interface

FORTH is described as an extensible language and for this to be true there must be some means of saving the extended language and reloading the new version with all the Cold Start values updated. (See Section 1.6 COLD START TABLES). FORTH is a disc-oriented language and ideally the regenerated language would be stored as a file on the machine's disc. On the absence of a disc, the next best thing is to use the cassette system, and this is what is done using the word SYS-SAVE. This updates the Cold Start Table values as required and invokes the nas-FORTH word WRITE, which in turn invokes the Nas-Sys routine "SCAL 'W'" which has the effect of executing the "W" command to create a cassette tape of the memory area that we wish to preserve.
(See Section 2.9 NAS-SYS CALLS).

1.4 Ram Disc Simulation

As mentioned in the last section, FORTH is a disc-oriented language and it's usual interface to the disc is by a virtual-memory system. Virtual-memory is a technique by which slow off-line storage (the Disc) is made to appear to the user as just another area of RAM, only slower. This means that the user does not have worry about files and other horrible things that only serve to make life more complex . Of course it is possible to write a disc operating system in FORTH that will support directories and such-like, but it is much more complex than the virtual memory system provided, and does not offer much except compatibility with files created by other languages.

The purpose of the disc is to store virtual-memory blocks, called "Screens" or "Blocks", which usually contain the source definitions of an application. Because FORTH is a compiler, the source is only used once and from then on the compiled code is run without the source. (In contrast to BASIC where the source is used all the time). Therefore FORTH is not self documenting and some means is required for mass-storage. The block concept is quite versatile, as it can be used to store messages and even compiled code as well as source text.

On a system such as the Nascom, where the possibility of discs at a later date exists, and where there is no shortage of RAM, it is much more sensible to simulate the disc in RAM than to abandon the virtual-memory concept and implement another way of storing source. On the Nascom system, tha base is set by the contents of the user variable LO. In order to examine the address held by LO, type HEX LO ? <cr>. The RAM-Disc Simulation area starts from the address held in LO and extends upwards. Therefore if you have a 48k RAM system, starting at 1000 hex, you would have 32k available for RAM-Disc simulation (32 pages). You must make sure that you do not exceed your RAM limits, as strange and nasty things could begin to happen. (See Section 1.7 EXPANDING nas-FORTH). When you have determined the top of your RAM-Disc area, store that address into the user variable HI. i.e, say, HEX 6000 HI !

A word called SCR-SAVE saves the RAM area between HI and LO on the tape, using the same method as described for SYS-SAVE. Of course a disc system will allow you a much larger number of screens (up to 500 or more) because the screens are stored on the disc and not as memory.

When you first list a screen, you will see all sorts of pretty patterns on the screen. This is because your RAM contains random values. In order to clear the screen type n CLEAR where n is the current screen number.

1.5 Memory Map

```
ADDRESS              DESCRIPTION
1000 Hex             Cold & Warm Start Vectors  = 0 +ORIGIN
  ^                  Implementation Attributes
  I                  Cold Start Tables
  v
1026                 Debug Support Area
  ^
  I
  v
103F                 Inner Interpreter (Address Threader)
  ^                       (Only 13 bytes long !)
  I
  v
104C                 Start of FORTH Dictionary
  ^
  I
  I
  I
  v
34A0                 Initial end of Dictionary
  I                  = Cold (DP) Value
  I                  = Cold (FENCE) Value
  I
  I  New Definitions
  v

  ^
  I  Parameter
  I  Stack
4B00                 Cold (SP) Value = (S0)
  I                  = (TIB)
  I  Terminal
  I  Input
  I  Buffer
  v


  ^
  I  Return
  I  Stack
4BA0                 Start of User Variables
  ^                  = Cold (RP) Value = (R0)
  I                  = (UP)
  I
  v                  End of User Variables
4BE0                 Start of Disc Buffers
  ^                  = FIRST
  I
  v
4FFF                 End of Disc Buffers
5000                 Last Memory location used + 1
                     = LIMIT
```

1.6 Cold Start Tables

When FORTH initially starts up there are a number of pointers and other memory locations (normally variables) which require setting with known values. These known values are stored in a cold start table, and can be altered (with care) by the user so as to change various characteristics or parameters of FORTH.

A FORTH word called +ORIGIN is used to reference the Cold Start area and when supplied with a suitable parameter to use as an offset will return the correct address of the cold start parameter. The usage of the cold start area is as follows:

8 +ORIGIN gives	1008	=	FIG Release number
9 +ORIGIN	1009	=	FIG Revision number
10 +ORIGIN	100A	=	User Version number
11 +ORIGIN	100B	=	Implementation Attributes
12 +ORIGIN	100C	=	Topmost word in FORTH dictionary. (altered by SYS-SAVE when saving extended system)
14 +ORIGIN	100E	=	Contains backspace character code for system. In nas-FORTH = 8 Hex
16 +ORIGIN	1010	=	Contains the initial user pointer UP. = 4BA0 Hex and can be altered with care.

The following values are used by the word COLD

18 +ORIGIN	1012	=	Contains the initial Parameter Stack Pointer value (S0). Is altered when expanding the system.
20 +ORIGIN	1014	=	Contains the initial Return Stack Pointer value. (R0). Is altered when expanding the system.
22 +ORIGIN	1016	=	Contains the initial Terminal Input Buffer (TIB) address. Is altered when expanding the system.
24 +ORIGIN	1018	=	Contains the initial value of the word WIDTH. (See Section 3.1 FORTH GLOSSARY). Initial value = 31
26 +ORIGIN	101A	=	Contains the initial value of the word WARNING. (See Section 3.0 FORTH GLOSSARY). Initial value = 0
28 +ORIGIN	101C	=	Contains the initial value of the word FENCE. (See Section 3.1 FORTH GLOSSARY).
30 +ORIGIN	101E	=	Contains the initial value for the dictionary pointer. This value changes every time a word is added to the dictionary.
32 +ORIGIN	1020	=	Contains the initial value of the word VOC-LINK. (See Section 3.0 FORTH GLOSSARY).

1.7 The User Area

The User Area contains the User Variables, and the User Area is pointed to by the contents of UP.

User Variables are not like ordinary variables. With an ordinary variable (one defined by the word VARIABLE), the value is kept in the parameter field of the dictionary entry. Each user variable, on the other hand, is kept in an array called the "User Area" The dictionary entry for each user variable is located elsewhere; it contains an offset into the user area. When you execute the name of a user variable, such as WIDTH, this offset is added to the beginning address of the user area. This gives you the address of WIDTH in the array, allowing you to use @ or ! in the normal way.

In the fig-FORTH model, there is only one user area, but on a multi-terminal or multi-tasking system such as poly-FORTH there are many user areas, one for each terminal or task. The advantage of this is that any number of tasks can use the same definition of a variable and each get its own value. Each task that executes

```
BASE @
```

gets the value for BASE from its own user area. This saves a lot of room in the system while still allowing each task to execute independently.

User variables are defined by the word USER (See Section 3.0 FORTH GLOSSARY). A number of user variables are already defined for use by the system, and offsets relating to these variables should not be used in new USER definitions.

System User Variables

Name	Offset	Purpose
S0	6	Parameter Stack Pointer
R0	8	Return Stack Pointer
TIB	10	Address of Terminal Input Buffer
WIDTH	12	Length of NFA
WARNING	14	See Section 3.1 FORTH GLOSSARY
FENCE	16	ditto
DP	18	Dictionary Pointer
VOC-LINK	20	See Section 3.1 FORTH GLOSSARY
BLK	22	ditto
IN	24	
OUT	26	

(Continued)

(Continued)

SCR	28	See Section 3.1 FORTH GLOSSARY
OFFSET	30	ditto
CONTEXT	32	
CURRENT	34	
STATE	36	
BASE	38	Current number base of system
DPL	40	See Section 3.1 FORTH GLOSSARY
FLD	42	ditto
CSP	44	
R£	46	
HLD	48	
'R/W	50	See Section 3.1 nas-FORTH EXTRA GLOSSARY
LO	52	ditto
HI	54	

Offsets above 54 decimal are therefore available for use with the defining word USER. Be careful that you do not exceed the space allocated for the user area (study the memory map!!).

1.8 Expanding nas-FORTH

Study of the memory map (See Section 1.5) will reveal that nas-FORTH is set up to run in 16k of RAM starting from 1000 Hex. Further study of the sections describing the user pointers and the user area, will reveal that locations exist which contain the various system pointers and that changes to these (the correct changes !!) will enable the user to change various features of his system. I might add that 16k of RAM is small by BASIC programming standards, but very large by FORTH standards. This is because of the compactness of the compiled form of word definitions.

Changes to the standard system fall into 3 groups:

1/ Increase the number of disc buffers.
(You will not normally need more that 2 disc buffers, nas-FORTH is set up to have 1 buffer.)

2/ Increase the amount of RAM available for word definitions (more dictionary space).
(This is normally only required for very large applications, or uses with many different Vocabularies).

3/ Increase the size of the Ram Disc Simulation area.

Further details on expanding nas-FORTH will appear in the User Group Newsletter.

1.9 Number Bases

You must remember that computers only work with binary patterns of predetermined voltage levels and therefore some sort of conversion process must be involved in order to present a reasonable interface to the user. In the final event all numerical input from the user must be converted to the binary values that the computer is able to use, and in the same way numerical output must be converted from the internal binary form in the computer to the user's chosen number base.

If you run BASIC, for example, you normally have no option but Decimal. In FORTH, however, you can use any number base you desire, providing you can represent that number set using the ASCII characters. (Try defining base 300 using ASCII characters !!!). When the FORTH word NUMBER performs conversion it goes and gets the contents of the user variable BASE and converts the number according to the value stored in BASE. So if BASE contains 2 then you are in BINARY. If BASE contains 10 then you are in DECIMAL. If BASE contains 16, HEX, and so on. Obviously, then, the definition of BINARY must be 2 BASE ! and HEX is 16 BASE !. If you wanted to use base 60 you could define a word called, say, SIXTY as : SIXTY 60 BASE ! ;

The number base can be changed dynamically (in other words, from within executing programs), and this is one of the most useful and powerful features of FORTH. Have fun!!

1.10 Error Messages

FORTH's usual method of dealing with a error situation is to examine the contents of the variable WARNING, and if it finds a 0, an error message number is issued. If it finds a non-zero value, it prints error text from Screen 4 & 5 using the word MESSAGE. Obviously, if you have a tape system, an error number is not very helpful. On nas-FORTH, the more common error numbers have been replaced by a helpful, descriptive message.

1	Empty Stack
2	Dictionary Full
3	Incorrect Address Mode
4	Isn't Unique
5	Unexpected End of Input
6	Range Error
7	Full Stack
8	-
9	Modifying Protected Address
10	Illegal Argument(s)
11	-
12	Array Out of Bounds
13	-
14	-
15	-
16	-
17	Compilation Only
18	Execution Only
19	Conditionals Not Paired
20	Definition Not Finished
21	In Protected Dictionary
22	Use Only When Loading
23	Off Current Editing Screen
24	Declare Vocabulary
25	Return Stack Not Balanced
26	Assembler Syntax Error
27	Relative Address Range Error

2.0 The Assembler

nas-FORTH can assemble machine-language definitions of words. Among the many examples of words defined by machine-language instructions are the operations :

+ - EXECUTE BRANCH SWAP DROP DUP

Such words are called code definitions and are constructed by the use of the resident assembler command CODE . The assembler is not intended for conventional programs. FORTH code routines are distinguished by the fact that all end by returning to the inner interpreter rather than by executing a conventional subroutine return.

Assembler code is, by definition, machine dependant. There are many characteristics of FORTH assemblers that are relatively constant across all the processors on which FORTH has been implemented. The main use of CODE definitions in FORTH is to interface to the operating system of the machine, to supply device drivers and to replace speed-conscious repetition in higher-level definitions.

Specific topics included in this section are :

2.1 CODE DEFINITIONS

The FORTH defining word CODE creates a standard dictionary entry whose code field address (CFA) contains the address of the byte that follows (the PFA), in other words the CFA points towards the PFA. (In a high-level definition the CFA points towards the word COLON, which is able to thread the following PFA's together, the PFA's of a high-level word being a list of addresses which in turn have the same structure, until a code definition is reached, which contains directly executable machine-code.) In a code definition the PFA is directly executable machine-code.

The form of a CODE definition is :

CODE NAME ... instructions ... code-ending END-CODE

CODE creates the definition, whose name is NAME. It also selects the ASSEMBLER vocabulary, in which the various instruction mnemonics, addressing modes, etc, are defined. These are used to build actual machine instructions, which are laid down in subsequent locations in the dictionary. CODE also saves the stack pointer position as the assembly of a definition commences. END-CODE checks that it has returned to the same position. In this way unfinished definitions can be detected. The code ending is one of several constants, all of which ultimately return to FORTH's address interpreter.

Aside from the dictionary header (NFA) there is no overhead in either space or time within a code definition. All instructions are executed at full machine speed.

2.2 CODE ENDINGS

Most FORTH code routines end with a jump to the address interpreter, sometimes after modifying the stack. Exceptions are interrupt routines (which return to the code that was being executed before the interrupt occurred) and routines which initiate processes whose completion will be signaled by an interrupt. These end with a jump to a special word.

The most common ending is NEXT JMP which simply is a jump to NEXT which is the entry point for the address interpreter. The word NEXT is defined as a constant which returns the address of NEXT in the inner interpreter.

In nas-FORTH, the normal stack is pointed to by the stack pointer register of the Z80, so a PUSH or POP of a register will move 16-bit values on or off the FORTH stack. An example would be the word DUP which is defined as :

```
CODE DUP   H POP   H PUSH   H PUSH   NEXT JMP   END-CODE
```

By reference to the FORTH glossary, DUP expects a parameter on the stack, and H POP takes this off the stack and into the HL register pair. H PUSH puts it back on the stack once and H PUSH again, puts a second copy on the stack. The words NEXT JMP allow the FORTH address threader to go on to the next word to be executed or return to the key-board. If this instruction is missing FORTH does not link properly and the system crashes ! The word END-CODE checks the stack and returns to the FORTH vocabulary.

There are two other FORTH words, also constants, called WHPUSH and HPUSH. They represent two more code ending words or entry points to the FORTH address interpreter. The definition above could be written more efficiently :

```
CODE DUP   H POP   H PUSH   HPUSH JMP   END-CODE
```

This saves 1 byte of code and is faster and more elegant. The word HPUSH JMP is equivalent to the sequence H PUSH NEXT JMP and the word WHPUSH is equivalent to W PUSH H PUSH NEXT JMP. The object of having three entry points to the threader is that it saves repetition of the sequences W PUSH, H PUSH in a number of code definitions and therefore is much more efficient over a number of definitions.

2.3 REGISTER USAGE

Registers in nas-FORTH are assigned as follows :

Register	ZILOG Name	FORTH Name	Usage
0	B	I	Address of the next location for the address interpreter (IP)
1	C	I'	
2	D	W	Address of the word being referenced; set by NEXT ; may be used as scratch.
3	E	W'	
4	H	H	Scratch ; H is a 16-bit register, with L as the low-order byte.
5	L	L	
6	M SP AF	M SP AF	
7	A	A	Accumulator (scratch)

Note

Some of these 8-bit registers are normally used in pairs. The upper and lower bytes are individually named, however, so that they may be used directly.

USE OF THE SYSTEM POINTERS

The register SP contains the address of the top byte of the normal (or parameter) stack. Any CODE words that manipulate the stack must be careful to leave SP pointing correctly. Usually this can be done conveniently by using the mnemonics POP and PUSH and branches to HPUSH and WHPUSH .

Register pair BC is used to hold the interpretive pointer (I) and must be preserved inside code definitions.

Register pairs I and W are used by the inner interpreter. The heart of the interpreter, at location NEXT is :

```
I LDAX   I INX   A L MOV   I LDAX   I INX   A H MOV
  M W' MOV   H INX   M W MOV
  XCHG   PCHL
```

Here it is assumed that I is pointing at an address compiled

into a colon definition. The first line transfers this address to register pair HL while advancing I . HL now contains the address of the code address of the word to be executed. The second line moves the code address itself to W , incrementing HL in the process. The XCHG in the third line interchanges H and W , so that now the code address plus 1 is in W . Finally, PCHL branches to the executable code. (Refer to the Glossary, under NEXT, for more information).

Normally you will not want to alter I , since this will alter program flow after returning to NEXT . W , however, transmits useful information, namely, the code address plus 1 of the word being executed. The starting address of the parameter field is one greater than this. For example, consider a definition of VARIABLE :

```
: VARIABLE      CREATE   2 ALLOT ;CODE   W INX   W PUSH
    NEXT JMP

VARIABLE  VAR
```

When VAR is invoked, the code following the ;CODE in the definition of VARIABLE will be executed. This code must put the address of the parameter field of VAR on the stack. This is done by incrementing W and PUSHing it onto the stack.

2.4 Assembler Instructions

To compile a colon definition the interpreter enters a special compile mode, in which the words of the input string are not executed (unless designated as IMMEDIATE). Instead their addresses are placed sequentially in the dictionary. During assembly, on the other hand, the interpreter remains in execute mode. The mnemonics of the assembler are defined as words which, when executed, assemble the corresponding operation code at the next location in the dictionary. As elsewhere in FORTH, operands (addresses or registers) precede operators (instruction mnemonics.)

Depending on the processor, several kinds of instuctions and addressing modes are possible. In the case of the Z80, where there is not a complete set of relative jumps, the relative mode will normally be selected where the condition allows it. If a forward jump is found to be out of range an error message is issued. For backward jumps there is no need for a range error as the mode can automatically switch to extended addressing.

The mnemonics are 8080 with Z80 extensions. Where appropriate the 8080 mnemonics have been generalised. Registers are defined as constants which enables registers to be passed more easily as arguments to MACROS.

Z80 ASSEMBLER INSTRUCTION GROUPS

8 BIT LOAD GROUP

```
s r MOV        Move the contents of s to r
n r MVI        Move the value n to r

          where s,r = A,I,I',W,W',H,L or M

  I LDAX       Load A with the contents of memory
  W LDAX       location pointed to by I or W

  I STAX       Store A in memory location pointed
  W STAX       to by I or W

 nn LDA        (nn) --> A
 nn STA        A --> (nn)

    IX+ r MOV d DISP        (IX+d) --> r
    r IX+ MOV d DISP        r --> (IX+d)
    n IX+ MOV d DISP        n --> (IX+d)

    IY+ r MOV d DISP        (IY+d) --> r
    r IX+ MOV d DISP        r --> (IY+d)
    n IX+ MOV d DISP        n --> (IX+d)

LDAR           Load A from refresh counter register
LDAI           Load A from interrupt vector register
STAR           Store A in refresh counter register
STAI           Store A in interrupt vector register
```

16 BIT LOAD GROUP

```
rr   PUSH          rr = AF, I, W, H, IX or IY
rr   POP
```

```
nn xx LXI    Load 16-bit register xx immediate. nn --> xx
nn xx LXD    Load 16-bit register xx direct   (nn) --> xx
nn xx SXD    Store 16-bit register xx direct  xx --> (nn)

             xx = I, W, H, IX or IY
```

```
xx LSPX      Load stack pointer SP from 16-bit register
                                              xx --> SP

             xx = H, IX or IY
```

EXCHANGE GROUP

EX	Exchange AF with AF'	AF <--> AF'.
EXX	Exchange BC with BC'	BC <--> BC'
	DE with DE'	DE <--> DE'
	HL with HL'	HL <--> HL'
XCHG	Exchange DE with HL	DE <--> HL
XTX	Exchange top of stack with index register.	xx <--> (SP)
	xx = H, IX or IY	
XTI	Exchange HL with top of stack.	HL <--> (SP)

BLOCK TRANSFER GROUP

LDI	Load and Increment	(HL) --> (DE) HL+1 --> HL DE+1 --> DE BC-1 --> BC
LDIR	Load, increment and repeat until BC=0	
LDD	Load and decrement	(HL) --> (DE) HL-1 --> HL DE-1 --> DE BC-1 --> BC
LDDR	Load, decrement and repeat until BC=0	

BLOCK SEARCH GROUP

CCI	Compare character and increment A=(HL)? HL+1 --> HL BC-1 --> BC
CCIR	Compare character and repeat until BC=0
CCD	Compare character and decrement A=(HL)? HL-1 --> HL BC-1 --> BC
CCDR	Compare character, decrement and repeat until BC=0

8-BIT ARITHMETIC & LOGIC

r ADD	A+r --> A
r ADC	A+r+c --> A
r SUB	A-r --> A
r SBC	A-r-c --> A
r AND	A and r --> A
r XOR	A xor r --> A
r OR	A or r --> A
r CP	Compare, & set flags according to A-r
r INR	r+1 --> r
r DCR	r-1 --> r

r = A, I, I', W, W', H, L, or M

IX or IY indexing is specified by IX+ or IY+ prior to the operation mnemonic, and d DISP after.

Example:

IX+ ADD d DISP = A+(IX+d) --> A

IMMEDIATE INSTRUCTIONS

n ADI	A+n --> A
n ACI	A+n+c --> A
n SUI	A-n --> A
n SBI	A-n-c --> A
n ANI	A and n --> A
n XRI	A xor n --> A
n ORI	A or n --> A
n CPI	A-n

GENERAL PURPOSE AF OPERATIONS

DAA	Decimal adjust A
CPL	Complement A
NEG	-A --> A
CCF	Complement carry flag
SCF	Set carry flag

16-BIT ARITHMETIC

```
rr DAD          HL+rr    --> HL
rr DADC         HL+rr+c  --> HL
rr DSBC         HL-rr-c  --> HL

rr = I, W, H or SP


rr DADIX        IX+rr    --> IX

rr = I, W, SP or IX


rr DADIY        IY+rr    --> IY

rr = I, W, SP or IY


rr INX          rr+1 --> rr
rr DCX          rr-1 --> rr

rr = I, W, HL, SP, IX or IY
```

ROTATE & SHIFT GROUP

RO	Rotate	SH	Shift
CIRC	Circular	AR	Arithmetic
VIA-C	Via carry flag	LO	Logical
RT	Right	LFT	Left
RRD	Rotate right decimal		
RLD	Rotate left decimal		

ROTATE INSTRUCTIONS SYNTAX

```
                  VIA-C     RT
      r    RO
                  CIRC      LFT
```

r = A, I, I', W, W', H, L, or M

```
IX+               VIA-C     RT
          RO                        d     DISP
IY+               CIRC      LFT
```

SHIFT INSTRUCTION SYNTAX

```
                  AR        RT
      r    SH
                  LO        LFT
```

r = A, I, I', W, W', H, L, or M

```
IX+               AR        RT
          SH                        d     DISP
IY+               LO        LFT
```

BIT MANIPULATION GROUP

```
b r  TBIT          Test bit b of r
b r  RBIT          Reset bit b of r
b r  SBIT          Set bit b of r

b    IX+  TBIT d   DISP     Test bit b of (IX+d)
b    IX+  RBIT d   DISP     Reset     ditto
b    IX+  SBIT d   DISP     Set       ditto
```

r = A, I, I', W, W', H, L, or M
b = 0, 1, 2, 3, 4, 5, 6 or 7 (bit number)

UNCONDITIONAL JUMPS, CALLS AND RETURNS

```
nn JMP          Jump to address nn
nn CALL         Call routine at address nn
H JP            HL --> PC
IX JP           IX --> PC
IY JP           IY --> PC
RET             Return   (SP) --> PC
RETI            Return from interrupt
RETN            Return from non-maskable interrupt
```

TEST CONDITIONS

UC	Unconditional
CS	Carry Set
CS NOT	Carry Clear
0=	Zero Flag Set
0= NOT	Zero Flag clear
PE	Parity even
PE NOT	Parity odd
0<	Negative (sign flag set)
0< NOT	Positive or Zero (Sign flag reset)

CONDITIONAL CALLS AND RETURNS

nn test	?CALL	Call nn if test true
test	?RET	Return if test true

The test can be any test condition except UC

RESTART GROUP

One byte calls to fixed locations

0	RST	Call to hex address	0000
1	RST		0008
2	RST		0010
3	RST		0018
4	RST		0020
5	RST		0028
6	RST		0030
7	RST		0038

INPUT AND OUTPUT GROUPS

n	IN	Input to A from port n
r	INP	Input to r from the port specified by the C register

r = A, I, I', W, W', H or L

INI	Input to (HL) from (C). HL+1 --> HL BC-1 --> BC
INIR	Input, increment and repeat.
IND	Input and decrement.
INDR	Input, decrement and repeat.

n	OUT	Output from A to port n
r	OUTP	Output from r to port (C)

OUTI	Output from (C) to (HL). HL+1 --> HL
OTIR	Output, increment and repeat.
OUTD	Output and decrement
OTDR	Output, decrement and repeat.

MISCELLANEOUS CPU CONTROL

NOP	No operation	
HALT	Halt	
DI	Disable interrupts	
EI	Enable interrupts	
IM0	Set interrupt mode 0	(8080 mode)
IM1	Set interrupt mode 1	(call to 38H)
IM3	Set interrupt mode 2	(Indirect call (using interrupt table and interrupt (register)
DJNZ	Decrement B register, jump if not zero.	

2.5 Macros

You may define macros easily in FORTH by using colon-definitions that contain assembler instructions.

For example:

```
ASSEMBLER  HEX     : LOOP       DBNZ UNTIL  ;

: TIMES    8 MVI   BEGIN   ;

: SRA      ( 16 Bit shift right arithmetic, of B, D, or H
                                          r count --- )
     TIMES   ROT DUP  SH AR RT  1+  RO  VIA-C RT LOOP ;
```

Macros are mainly a notational convenience; SRA is merely the sequence as described in the definition, just as if you had written the expression out in full.

The words used to implement the assembler structures (loops and conditionals) are defined as macros, as are the code endings.

2.6 Logical Structures

Control of logical flow is handled by FORTH's assembler using the same structured approach as high-level FORTH, although the implementation of the commands is necessarily different. The commands even have the same names as their high-level analogues; ambiguity is prevented by the use of separate vocabularies. The following are implemented as standard macros:

BEGIN	Puts an address on the stack (HERE)
UNTIL (or END)	Assembles a conditional jump back to the address left by BEGIN. It is preceded by a condition code. The loop is ended if the condition is met. Common condition codes are 0= and 0<, as appropriate to the various CPUs.
AGAIN	Assembles an unconditional jump back to the address left by BEGIN. Once you have entered a BEGIN - AGAIN loop there is no exit - it is infinite!!
NOT	Negates condition code.
IF	Assembles a conditional forward jump, to be taken if the preceding condition is false, leaving the address of this instruction on the stack.
ELSE	Provides the destination of IF's jump (whose address was on the stack) and assembles an unconditional forward jump (whose location is left on the stack).
THEN	Provides the destination for a jump instruction whose location is on the stack at assembly time (left by IF or ELSE).

The ELSE clause may be omitted entirely. This construction is functionally analogous to the IF ... ELSE ... THEN construction provided by FORTH's compiler. Since the locations or destinations of branches are left on the stack at assembly time, the structures may be nested naturally. By manipulating the stack during assembly, however, you can assemble any branching structure.

If you wish to branch forward, use IF to leave the location of the branch on the stack. At the branch's destination, bring the location back to the top of the stack (if it is not there already) and use ELSE or THEN to complete the branch by filling in its destination at the location that is on the top of the stack.

If you wish to branch back to an address, leave it on the stack with BEGIN. At the branch's source, bring the address to the top of the stack and use UNTIL or END or a jump mnemonic to assemble a conditional or unconditional branch back. Be sure to manipulate the branch address before the condition mnemonic since each condition code adds one item to the stack.

nas-FORTH Assembler has LABEL's but we do not recommend them in structured programming. Their functions are better performed by the words IF , ELSE , THEN , BEGIN , UNTIL , etc. Since CODE definitions are usually extremely short, labels are not particularly desirable; they tend to encourage complicated flow patterns that are not appropriate in FORTH.

2.7 Literals

Some processors allow you to define instructions to reference literals. (6502 for example). For these, the standard FORTH word for identifying a literal is £ . Thus the instruction:

```
1000 £ 0 MOV
```

would move the literal 1000 into register 0. A few processors allow a short instruction format for small literals and a long format for larger ones. In such cases the FORTH assembler automatically examines the literal and generates the appropriate format.

nas-FORTH uses a much simpler method, and literals are determined by the context of the instuction, and taken off the top of the stack. This is a much easier method and it is also the most natural. For example :

```
100 A MVI
```

which loads the A register with the literal 100. Another example is the 16-bit load :

```
1000 W LXI
```

which will load the W register (DE) with 1000.

2.8 Device Handlers

Device handlers should be kept extremely short, including only the instructions required to pass a value to or from the stack or to issue a command. Consider, for example, a self-scan character display that is interfaced to a RCA 1802 as Device 2. This is all that is needed to output one character from the top of the stack:

```
CODE DISPLAY  S INC   S SEX   2 OUT   NEXT JMP
                                END-CODE
```

In this example, S INC increments the stack pointer (to get out the low-order byte), S SEX sets S as the output register, and 2 OUT sends the character to the device, incrementing S again to complete a POP.

Given this hypothetical definition of DISPLAY, you could define TYPE at high level to display a string of characters whose byte address and length are on the stack:

```
: TYPE   0 DO  DUP C@ DISPLAY  1+ LOOP  DROP ;
```

To convert and display a number on the stack, you could define PRINT as:

```
: PRINT   (.) TYPE ;
```

Here (.) performs the conversion, leaving the address and length of the resulting string for TYPE. The point here is that, given the simple code definition DISPLAY, you have easy, full control of the display in high-level FORTH.

Device drivers are highly variable in nature, depending upon both the processor and the actual device.

2.9 Nas-Sys Calls

There is nothing special in performing calls to external subroutines, provided some rules are followed. Nas-Sys calls are quite easy because of the high standard of documentation provided by NASCOM. As an example, this is the actual definition of the word WRITE which invokes the Nas-Sys Write command.

```
HEX
     CODE WRITE   H POP   0C0E H SXD   H POP   0C0C H SXD
                I PUSH   3 RST   57 C,   I POP   JMP NEXT
     END-CODE
```

We will now explain why and what the FORTH assembler does with the definition above.

Assembly of WRITE on the Z80/8080

Instruction	Action During Assembly	During Execution
H	Pushes 4 (Register H) onto the stack	Pops the top of the stack into
POP	Assembles a POP instruction, Referencing register H	register HL
0C0E	Push value onto stack as a 16 bit literal. 0C0E = ARG2	Store the value in HL into
H	Pushes 4 (Register H) onto the stack	memory address ARG2 (0C0E)
SXD	Assemble a LD (nn),HL instruction, using the two stack values supplied.	
H		Pops the second parameter off
POP		the stack and into HL
0C0C		Puts HL value
H		into ARG1
SXD		(0C0C)
I	Pushes 0 (Register B) onto the stack	Saves the I reg so that Nas-sys
PUSH	Assembles a PUSH instruction, referencing Register BC	does not destroy it.
3	Pushes 3 onto the stack	Performs a jump to Nas-Sys
RST	Assembles a RST 18 instruction.	

(continued)

57	Pushes 57 onto the stack.	With the RST is the same as a SCAL 'W'
C,	Compiles the stack value into the dictionary as a literal.	
I POP		Restore the I reg.
NEXT	Pushes the address of the inner interpreter entry point onto the stack.	
JMP	Assembles a jump instruction, referencing the address on the stack.	Jump back to the address threader of FORTH.

3.1 Forth Glossary

This glossary contains all of the word definitions in Release 1.1 of fig-FORTH. The definitions are presented in the order of their ASCII sort.

The first line of each entry shows a symbolic description of the action of the proceedure on the parameter stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8 bit byte (i.e. high 8 bits zero)
c	7 bit ASCII character (high 9 bits zero)
d	32 bit signed double integer, most significant portion with sign on top of the stack.
f	boolean flag. 0 = false, non-zero = true
ff	boolean false flag = 0
tf	boolean true flag = non-zero
n	16 bit signed integer number
u	16 bit unsigned integer

The capital letters on the right show definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used, if greater than one.
E	Intended for execution only.
P	Has precedence bit set. Will execute even when compiling.
U	A User variable.

Unless otherwise noted, all references to numbers are for 16 bit signed integers. On 8 bit data bus computers, the high byte of a number is on top of the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16 bit signed integer math, with error and underflow indication unspecified.

! n addr ---
Store 16 bits of n at address. Pronounced "store"

!CSP Save the stack position in CSP. Used as part of the compiler security.

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

£> d --- addr count
Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE

£S d1 --- d2
Generates ASCII text in the text output buffer, by the use of £, until a zero double number n2 results. Used between <£ and £>

' --- addr P
Used in the form:
' nnnn
Leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in a colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an error message is given. Pronounced "tick"

(Used in the form: P
(cccc)
Ignore a comment that will be delimited by a right parenthesis on the same line. May occur during execution or in a colon-definition. A blank after the leading parenthesis is required.

(.") C+
The run-time proceedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(;CODE) C

The run-time proceedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the following machine code sequence. See ;CODE

(+LOOP) n --- C2

The run-time proceedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP

(ABORT) Executes after an error when WARNING is -1. This word normally executes ABORT, but may be altered (with care) to a user's alternative proceedure.

(DO) C

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

(FIND) addr1 addr2 --- pfa b tf (ok)
addr1 addr2 --- ff (bad)

Searches the dictionary starting at the name field address addr2, matching the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.

(LINE) n1 n2 --- addr count

Convert the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 64 indicates the full line text length.

(LOOP) C2

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

(NUMBER) d1 addr1 --- d2 addr2

Convert the ASCII text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. Addr2 is the address of the first unconvertable digit. Used by NUMBER.

* n1 n2 --- prod

Leave the signed product of two signed numbers

*/ n1 n2 n3 --- n4

Leave the ratio n4 = n1 * n2 / n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:

n1 n2 * n3 /

*/MOD n1 n2 n3 --- n4 n5

Leave the quotient n5 and remainder n4 of the operation n1 * n2 / n3

A 31 bit intermediate product is used as for */

+ n1 n2 --- sum

Leave the sum of n1 + n2

+! n addr ---

Add n to the value at the address. Pronounced "plus-store"

+- n1 n2 --- n3

Apply the sign of n2 to n1, which is left as n3

+BUF addr1 --- addr2 f

Advance the disc buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by variable PREV

+LOOP n1 --- (run)

addr n2 --- (compile) P,C2

Used in a colon-definition in the form:

DO ... n1 +LOOP

At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile-time error checking.

+ORIGIN n --- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, n ---
Store n into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced "comma"

- n1 n2 --- diff
Leave the difference of n1 - n2

--> Continue interpretation with the next disc screen. Pronounced "next screen"

-DUP n1 --- n1 (if zero)
n1 --- n1 n1 (non-zero)
Reproduce n1 only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND --- pfa b tf (found)
--- ff (not found)
Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then the CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-TRAILING addr n1 --- addr n2
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. i.e. the characters at addr+n1 to addr+n2 are blanks.

Print a number from a signed 16 bit two's complement value, converted according to the numeric BASE. A trailing blank follows. Pronounced "dot"

Used in the form:

." cccc"

Compiles an in-line string cccc (delimited by the trailing ") with an execution proceedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final " . The maximum number of characters may be installation dependant. See (.")

.LINE line scr ---

Print on the terminal device, a line of text from the disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---

Print the number n1 right aligned in a field whose width is n2. No following blank is printed.

/ n1 n2 --- quot

Leave the signed quotient of n1 / n2

/MOD n1 n2 --- rem quot

Leave the remainder and signed quotient of n1 / n2. The remainder has the sign of the dividend.

0 1 2 3 --- n

These small numbers are used so often that it is attractive to define them by name in the dictionary as constants.

0< n --- f

Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= n --- f

Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0BRANCH f --- C2

The run-time proceedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n1 --- n2
Increment n1 by 1. Pronounced "one plus"

2+ n1 --- n2
Increment n1 by 2. Pronounced "two plus"

: P,E
Used in the form called a colon-definition:
: cccc .... ;
Creates a dictionary entry defining cccc as equivalent to the following sequence of Forth word definitions '....' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as state is non-zero. Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

; P,C
Terminate a colon-definition and stop further compilation. Compiles the run-time routine ;S

;CODE P,C
Used in the form:
: cccc .... ;CODE
assembly mnemonics
Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following mnemonics.

When cccc later executes in the form:
cccc nnnn
the word nnnn will be created with its execution proceedure given by the machine code following cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn. An existing defining word must exist in cccc prior to ;CODE.

;S P
Stop interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling proceedure.

< n1 n2 --- f
Leave a true flag if n1 is less than n2 ; otherwise leave a false flag.

<£ Set-up for pictured numeric output farmatting using the words:

```
<£  £  £S  SIGN  £>
```

The conversion is done on a double number producing text at PAD.

<BUILDS C

Used within a colon-definition:

```
: cccc   <BUILDS    ....
         DOES>      ....      ;
```

Each time cccc is executed, <BUILDS defines a new word with a high-level execution proceedure. Executing cccc in the form:

```
cccc  nnnn
```

uses <BUILDS to create a dictionary entry for nnnn with a call to the DOES> part for nnnn. When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in cccc. <BUILDS and DOES> allow run-time proceedures to be written in high-level rather than in assembler code (as required by ;CODE).

= n1 n2 --- f

Leave a true flag if n1 = n2 ; otherwise leave a false flag.

> n1 n2 --- f

Leave a true flag if n1 is greater than n2 ; otherwise a false flag.

>R n --- C

Remove a number from the parameter stack and place as the most accessable on the return stack. Use should be balanced with R> in the same definition. Pronounced "to R"

? addr ---

Print the value contained at the address in free format according to the current BASE.

?COMP Issue error message if not compiling.

?CSP Issue error message if stack position differs from value saved in CSP.

?ERROR f n ---
Issue an error message number n, if the boolean flag is true.

?EXEC
Issue an error message if not executing.

?LOADING
Issue an error message if not loading.

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

?STACK
Issue an error message if the stack is out of bounds. This definition may be installation dependant.

?TERMINAL --- f
Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation. This definition is installation dependant.

@ addr --- n
Leave the 16 bit contents of address addr as n. Pronounced "fetch"

ABORT
Clear the stacks and enter the execution state. Return control to the operator's terminal, printing a message appropriate to the installation.

ABS n --- u
Leave the absolute value of n as u.

AGAIN addr n --- (compiling) P,C2
Used in a colon-definition in the form:
BEGIN .... AGAIN
At run-time, AGAIN forces execution to return to the corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

At compile-time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n ---

Add the signed number to the dictionary pointer DP. May be used to reserve dictionary space or re-origin memory. n is with regard to computer address type (byte or word).

AND n1 n2 --- n3

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

B/BUF --- n

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

B/SCR --- n

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

BACK addr ---

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

BASE --- addr U

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

BEGIN --- addr n (compiling) P

Occurs in a colon-definition in the form:

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

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile-time BEGIN leaves its return address and n for compiler error checking.

BL --- c

A constant that leaves the ASCII value for "blank".

BLANKS addr count ---
Fill an area of memory beginning at addr with blanks.

BLK --- addr U
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK n --- addr
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disc before block n is read into the buffer. See also BUFFER, R/W, UPDATE FLUSH.

BLOCK-READ
BLOCK-WRITE
These are the preferred names for the installation dependent code to read and write one block to the disc.

BRANCH C2
The run-time proceedure to unconditionally branch. an in-line offset is added to the interpretive pointer I to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.

BUFFER n --- addr
Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc. The address left is the first cell within the buffer for data storage.

C! b addr ---
Store 8 bits at address. On word addressing computers, further specification is necessary regarding byte addressing. Pronounced "c-store"

C, b ---
Store 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This is only available on byte addressing computers, and should be used with caution on byte addressing mini-computers. "c-comma"

C@ addr --- b

Leave the 8 bit contents of memory address addr. On word addressing computers, further specification is needed regarding byte addressing. Pronounced "c-at"

CFA pfa --- cfa

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

CMOVE from to count ---

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

COLD

The cold start proceedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.

COMPILE C2

When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

CONSTANT n ---

A defining word used in the form:

n CONSTANT cccc

to create word cccc, with its parameter field containing n. When cccc is later executed, it will push the value of n onto the stack.

CONTEXT --- addr U

A user variable containing a pointer to the vocabulary within which dictionary searches will first begin.

COUNT addr1 --- addr2 n

Leave the byte address addr2 and byte count n of a message text beginning at address addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts with the second byte. Typically COUNT is followed by TYPE.

CR

Transmit a carriage return and line feed to the selected output device.

CREATE

A defining word used in the form:

CREATE cccc

by such words as CODE and CONSTANT to create a dictionary header for a Forth definition. The code field contains the address of the word's parameter field. The new word is created in the CURRENT vocabulary.

CSP

--- addr U

A user variable temporarily storing the stack pointer position, for compilation error checking.

CURRENT

--- addr U

A user variable containing a pointer to the vocabulary into which new definitions will be placed. As soon as a definition is made in the CURRENT vocabulary, it automatically becomes also the context vocabulary.

D+

d1 d2 --- dsum

Leave the double number sum of two double numbers.

D+-

d1 n --- d2

Apply the sign of n to the double number d1, leaving it as d2.

D.

d ---

Print a signed double number from a 32 bit two's complement value. The high order 16 bits are on top of the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced "d-dot"

D.R

d n ---

Print a signed double number d right aligned in a field n characters wide.

DABS

d --- ud

Leave the absolute value ud of a double number d.

DECIMAL

Set the numeric conversion BASE for decimal input-output.

DEFINITIONS

Used in the form:

```
cccc  DEFINITIONS
```

Set the CURRENT vocabulary to the CONTEXT vocabulary. In the example, executing vocabulary name cccc made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary cccc.

DIGIT

```
c   n1   ---  n2   tf   (ok)
c   n1   ---  ff        (bad)
```

Converts the ASCII character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.

DLITERAL

```
d    ---  d          (executing)
d    ---             (compiling)          R
```

If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

DMINUS

```
d1   ---  d2
```

Convert d1 to its double number two's complement.

DO

```
n1   n2   ---        (execute)
addr  n   ---        (compile)          P,C2
```

Occurs in a colon-definition in the form:

```
DO  ....  LOOP
DO  ....  +LOOP
```

At run-time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO ; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop 'I' will copy the current value of the index to the stack. See I, LOOP, +LOOP, LEAVE.

When compiling within the colon-definition, DO compiles (DO), leaves the following address addr and n for later error checking.

DOES>

A word which defines the run-time action within a high-level defining word. DOES> alters the code field and first parameter of the new word to

execute the sequence of compiled word addresses following DOES>. Used in combination with <BUILDS. When the DOES> part executes it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the Forth assembler, multi-dimensional arrays, and compiler generation.

DP --- addr U

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

DPL --- addr U

A user variable containing the number of digits to the right of the decimal on double integer input. It may also be used to hold output column location of a decimal point, in user generated formatting. The default value on single number input is -1.

DR0
DR1

Installation dependant commands to select disc drives, by presetting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection. OFFSET is suppressed for error text so that it may always originate from Drive 0.

DROP n ---

Drop the number from the stack.

DUMP addr n ---

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

DUP n --- n n

Duplicate the value n on the stack.

ELSE addr1 n1 --- addr2 n2
(compiling) P,C2

Occurs within a colon-definition in the form:

```
IF .... ELSE .... ENDIF
```

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time ELSE emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing at addr1.

EMIT c ---

Transmit ASCII character c to the selected output device. OUT is incremented for each character output.

* In nas-FORTH the eighth bit is not suppressed, so that EMIT can operate the graphics ROM if you have one fitted. *

EMPTY-BUFFERS Mark all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disc. This is also an initialisation proceedure before first use of the disc.

ENCLOSE addr1 c --- addr1 n1 n2 n3

The text scanning primitive used by WORD. From the text address addr1 and an ASCII delimiting character c, is determined the byte offset to the first non-delimiting character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included. This proceedure will not process past an ASCII 'null', treating it as an unconditional delimiter.

END P,C2

This is an 'alias' or duplicate definition for until.

ENDIF addr n --- (compile) P

Occurs in a colon-definition in the form:

```
IF .... ENDIF
IF .... ELSE .... ENDIF
```

At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also IF and ELSE.

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

ERASE addr n ---
Clear a region of memory to zero from addr over n addresses.

ERROR line --- in blk
Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 4 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 4. If WARNING = 0, n is just printed as a message number (non-disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. The final word executed in ERROR is QUIT.

EXECUTE addr ---
Execute the definition whose code field address is on the stack. The code field address is known as the CFA or compilation address.

EXPECT addr count ---
Transfer characters from the terminal to address, until a "return" or the count of characters have been received. One or more nulls are added at the end of the text.

FENCE --- addr U
A user variable containing an address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE.

FILL addr quan b ---
Fill memory at the address with the specified quantity of bytes b.

FIRST --- n
A constant that leaves the address of the first (lowest) block buffer.

FLD --- addr U
A user variable for control of number output field width. Presently un-used in fig-FORTH.

FORGET E

Executed in the form:
FORGET cccc
Deletes definition named cccc from the dictionary with all entries physically following it. In fig-FORTH, an error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

FORTH P

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. Until additional user vocabularies are defined, new user definitions become a part of FORTH. FORTH is immediate, so it will execute during the creation of a colon-definition, to select this vocabulary at compile-time.

HERE --- addr
Leave the address of the next available dictionary location.

HEX Set the numeric conversion base to sixteen (hexadecimal).

HLD --- addr
A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c ---
Used between <£ and £> to insert an ASCII character into a pictured numeric output string.
e.g. 2E HOLD will place a decimal point.

I --- n C

Used within a DO-LOOP to copy the loop index to the stack. Other use is implementation dependant. See R

ID. addr ---
Print a definition's name from its name field address. Pronounced "I-Dee-Dot"

IF

f --- (run-time)
--- addr n (compiling) P,C2

Occurs in a colon-definition in the form:

IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ... ENDIF

At run-time, IF selects execution based on a boolean flag. If f is true (non-zero), execution continues ahead through the true part (tp). If f is false (zero), execution skips till just after ELSE to execute the false part (fp). After either part, execution resumes after ENDIF. ELSE and its false part are optional.; if missing, false execution skips to just after ENDIF.

At compile-time IF compiles 0BRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE

Mark the most recently made definition so that when encountered at compile-time, it will be executed rather than being compiled. i.e. the precedence bit in its header is set. This method allows definitions to handle unusual compiling situations, rather than build them into the fundamental compiler. The user may force compilation of a immediate definition by preceeding it with the word [COMPILE].

IN

--- addr U

A user variable containing the byte offset within the current input text buffer (terminal or disc) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX

from to ---

Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disc screens.

INTERPRET

The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disc) depending on STATE. If the word name cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with a "?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than

to force this action. See NUMBER

KEY --- c
Leave the ASCII value of the next terminal key struck.

LATEST --- addr
Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE C
Force termination of a DO-LOOP at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa --- lfa
Convert the parameter field address of a dictionary definition to its link field address.

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

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

LIT C2
Within a colon-definition, LIT is automatically compiled before each 16 bit literal number encountered in the input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n --- (compiling) P,C2
If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon-definition. The intended use is:

```
: xxx [ calculate ] LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and LITERAL compiles this value.

LOAD n ---

Begin interpretation of screen n. Loading will terminate at the end of the screen or at ;S. See ;S and -->

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

Occurs in a colon-definition in the form:

```
DO .... LOOP
```

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

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

M* n1 n2 --- d

A mixed magnitude math operation which leaves the double number signed product of two signed numbers. Pronounced "M-star"

M/ d n1 --- n2 n3

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

M/MOD ud2 u2 --- u3 u4

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2. Pronounced "M-slash-mod"

MAX n1 n2 --- max

Leave the greater of two numbers, n1 and n2

MESSAGE n ---

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

MIN n1 n2 --- min
Leave the smaller of two numbers, n1 and n2

MINUS n1 --- n2
Leave the two's complement of n1

MOD n1 n2 --- mod
Leave the remainder of n1/n2, with the same sign as n1.

MOVE addr1 addr2 n ---
Move the contents of n memory cells (16 bit contents) beginning at addr2. The contents of addr1 are moved first.

NEXT This is the inner interpreter that uses the interpretive pointer (I) to execute compiled FORTH definitions. It is not directly executed but is the return point for all code proceedures. It acts bt fetching the address pointed to by I, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of FORTH. Locations of I and W are computer specific.

* In nas-FORTH, NEXT is defined as a constant that pushes the address of the entry point to the inner interpreter onto the stack. This is very useful in code definitions. See nas-FORTH Assembler Features section *

NFA pfa --- nfa
Convert the parameter field address of a definition to its name field.

NUMBER addr --- d
Convert a character string left at addr with a preceeding count, to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

OFFSET --- addr U

A user variable which may contain a block offset to disc drives. The contents of OFFSET are added to the stack number by BLOCK. Messages by MESSAGE are independant of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

OR n1 n2 -- or

Leave the bit-wise logical or of two 16 bit values n1 and n2.

OUT --- addr U

A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.

OVER n1 n2 --- n1 n2 n1

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

PAD --- addr

Leave the address of the text output buffer, which is a fixed offset above HERE.

PFA nfa --- pfa

Convert the name field address of a compiled definition to its parameter field address.

PREV --- addr

A variable containing the address of the disc buffer most recently referenced. The UPDATE command marks this buffer to be later written to disc.

QUERY

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

QUIT

Clear the return stack, stop compilation, and return control to the operator's terminal. No message is given.

R --- n

Copy the top of the return stack to the parameter stack.

R£ --- addr U

A user variable which may contain the location of an editing cursor, or other file related function.

R/W addr blk f ---

The fig-FORTH standard disc read-write linkage. addr specifies the source or destination block buffer, blk is the sequential number of the referenced block; and f is a flag for f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking.

* In nas-FORTH, R/W is vectored to allow flexibility. R/W fetches the address of the RAM-Disc simulation word from the variable 'R/W and executes it. The effect is the same, but R/W can be redirected by storing another word's address into 'R/W. Used in this way it is a crude way of resolving forward references. *

R> --- n

Remove the top value from the return stack and leave it on the parameter stack. Pronounced "From-R" See >R and R

R0 --- addr U

A user variable containing the initial location of the return stack. Pronounced "R-zero" See RP!

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

Used within a colon-definition in the form:

BEGIN .... WHILE .... REPEAT

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

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

ROT n1 n2 n3 --- n2 n3 n1

Rotate the top three values on the stack, bringing the third to the top.

RP! A computer dependent proceedure to initialise the return stack pointer from user variable R0. Pronounced "R-P-Store"

S->D n --- d
Sign extend a single number to form a double number.

S0 --- addr U
A user variable that contains the initial value for the stack pointer. Pronounced "S-zero" See SP!

SCR --- addr U
A user variable containing the screen number most recently referenced by LIST.

SIGN n d --- d
Stores an ASCII "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <£ and £>.

SMUDGE
Used during word definition to toggle the "smudge bit" in a definition's name field. This prevents an un-completed definition from being found during dictionary searches, until compiling is completed without error.

SP!
A computer dependent proceedure to initialise the stack pointer from user variable S0. Pronounced "S-P-Store"

SP@ --- addr
A computer dependent proceedure to return the address of the stack position to the top of the stack, as it was before SP@ was executed.
(e.g. 1 2 SP@ . . . would type 2 2 1)

SPACE
Transmit an ASCII blank to the output device.

SPACES n ---
Transmit n ASCII blanks to the output device.

STATE --- addr U
A user variable containing the compilation state. A non-zero value indicates compilation. The value itself may be implementation dependent.

SWAP n1 n2 --- n2 n1
Exchange the top two values on the stack.

TASK
A no-operation word which can mark the boundary between applications.

THEN P,C0
An alias for ENDIF

TIB --- addr U
A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---
Complement the contents of addr by the bit pattern b.

TRAVERSE addr1 n --- addr2
Move across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward high memory; if n=-1, the motion is toward low memory. The addr2 resulting is the address of the other end of the name.

TRIAD scr ---
Display on the selected output device the three screens which include that numbered scr, begining with a screen evenly divisible by three. Output is suitable for source text records.

TYPE addr count ---
Transmit count characters from addr to the selected output device.

U* u1 u2 --- ud
Leave the unsigned double number product of two unsigned numbers. Pronounced "U-Star"

U/ ud u1 --- u2 u3
Leave the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1. Pronounced "U-Slash"

UNTIL

f --- (run-time)
addr n --- (compile-time) P,C2

Occurs within a colon-definition in the form:
BEGIN .... UNTIL
At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if f is true, execution continues ahead.

At compile-time, UNTIL compiles (0BRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE

Marks the most recently referenced block (pointed to by PREV) as altered. The block will subsequently be transferred automatically to disc should it's buffer be required for storage of a different block.

USE

--- addr
A variable containing the address of the block buffer to use next, as the least recently written.

USER

n ---
A defining word used in the form:
n USER cccc
which creates a user variable cccc. The parameter field of cccc contains n as a fixed offset relative to the user pointer UP for this user variable. When cccc is executed, it places the sum of it's offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE E

A defining word used in the form:
n VARIABLE cccc
When VARIABLE is executed, it creates the definition cccc with its parameter field initialised to n. When cccc is later executed, the address of it's parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VOC-LINK

--- addr U
A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting through multiple vocabularies.

VOCABULARY E

A defining word used in the form:

VOCABULARY cccc

to create a vocabulary definition cccc. Subsequent use of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.

In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined. All vocabularies ultimately chain to Forth. By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK

VLIST List the names of the definitions in the CONTEXT vocabulary.

WARNING --- addr U

A user variable containing a value controlling messages. If = 1 disc is present, and screen 4 of drive 0 is the base location for messages. If = 0, no disc is present and messages will be presented by number. If = -1, execute (ABORT) for a user specified proceedure. See MESSAGE, ERROR

WHILE f --- (run-time)
addr n1 --- ad1 n1 ad2 n2 P,C2

Occurs in a colon-definition in the form:

BEGIN .... WHILE (tp) ... REPEAT

At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part through to REPEAT, which the branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile-time, WHILE emplaces (0BRANCH) and leaves ad2 of the reserved offset. The stack values will be resolved by REPEAT.

WIDTH --- addr U

In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 through 31, with a default value of 31. The name, character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD c ---

Read the input text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string begining at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurances of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disc block stored in BLK. See BLK, IN.

X

This is the psuedonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the execution proceedure to terminate interpretation of a line of text from the terminal or within a disc buffer, as both buffers always have a null at the end.

XOR n1 n2 --- xor

Leave the bit-wise logical exclusive-or or the two values n1 and n2.

[P

Used in a colon-definition in the form:

```
: xxx  [  words  ]  more words ;
```

Suspend compilation. The words after [are executed, not compiled. This allows for calculation or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE] P,C

Used in a colon-definition in the form:

```
: xxx  [COMPILE] FORTH ;
```

[COMPILE] will force the compilation of an immediate definition, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxx executes, rather than at compile-time.

] Resume compilation, to the completion of a colon-definition. See [.

3.2 Forth Glossary

'R/W --- addr U

A user variable containing the address of the word MEMRW (a headerless word), which is the word that performs the RAM Disc Simulation. This variable is used for 'vectored execution' from the word R/W so that the address of a disc handler word can be placed into 'R/W in order to add discs at a later date.

.S This word will produce a stack display for debugging and other purposes. It is incorporated in the definition of the word QUIT for the automatic stack display feature of nas-FORTH.

BINARY Set the numeric conversion base for Binary input-output.

BYE Perform a jump to 00hex.

FREE Displays the amount of free memory in the current number base.

HI --- addr U

A user variable containing the top address of the Ram Disc Simulation area. Can be altered with care!!

HPUSH --- n

A constant which puts the address of a re-entry point to the inner interpreter on the top of the stack. (See Section 2.0 The Assembler)

LO --- addr U

A user variable containing the bottom address of the RAM Disc Simulation area. Can be altered with care!!

PAUSE --- f

This word can be used instead of ?TERMINAL to give a friendlier response to a keyboard interruption. It scans the keyboard and when any key (apart from ESC) is hit, it enters an internal idle loop thereby stopping any further output. If ESC is hit, PAUSE executes QUIT and returns to the system

thereby terminating output. If any key other than ESC is hit, it carries on with the output.

SCR-SAVE

A word to save the contents of the Ram Disc Simulation area to the cassette recorder. This word invokes the Nas-sys 'Write' routine via the nas-FORTH word WRITE and uses the contents of the variables LO and HI to supply the source and destination addresses.

SYS-SAVE

This word is used to save a customised version of nas-FORTH, and it also uses the word WRITE. Before the word WRITE is used, a number of cold start table values are updated, and then the memory image of the new nas-FORTH is saved from 0 +ORIGIN to HERE.

WHPUSH

--- n

A constant which puts the address of a re-entry point to the inner interpreter on the top of the stack. (See Section 2.0 The Assembler)

WRITE

from to ---

This word invokes the Nas-sys routine 'Write', and takes the two values on the stack as starting and ending addresses. (See Section 2.9 Nas-sys Calls)

P!

b n ---

Sends byte b to I/O mapped port number n.
"Pee-Store"

P@

n --- b

Reads I/O mapped port number n and places byte b on the stack. "Pee-Fetch".

4.1 xFORTH User Group

The purchase price of nas-FORTH includes a year's free membership of the xFORTH User Group. This group is organised by David Husband and covers a wide variety of machines, apart from machines covered by David Husband's range of FORTH implementations.

The purpose of the group is to provide you with the support which you deserve when you purchase FORTH software, and to provide a forum for members to exchange ideas and information. A bulletin-board system is planned and the hardware for this system is well advanced.

The membership of the group is comprised of individuals, Universities, Government bodies, and large companies the common factor being that they are all FORTH users. Many are newcomers to FORTH and they are enthusiastic converts !

If you purchased your software direct from David Husband, then you will already be registered with the group, otherwise a form is included for you the send to us.

4.2 Any Problems ?

We are very anxious to ensure that you are satisfied with your FORTH software, so we hope you will feel free to contact us should you have any problems or queries at all.

We would prefer you to ring us on Bournemouth (0202) 764724 between the hours of 6pm and 7pm, Monday to Saturday so as to allow our work during the day to continue un-interrupted.

If you contact us by letter, we are not able to respond by letter in return, but your query would be answered in the next newsletter of the Users Group. We would like letters from users for inclusion in the newsletter in any event, whether you have a problem or not.

4.3 Acknowledgements

This version of FORTH is based on the public domain publications provided through the courtesy of the

FORTH Interest Group
P.O. Box 1105
San Carlos
California
CA. 94070

The nas-FORTH Assembler is based on an assembler written by Bill Stoddart of the Forth Interest Group UK

FIG UK can be found at

c/o Honorary Secretary
15 St Albans Mansion
Kensington Court Place
LONDON. W8 5QH

They publish a newsletter and make available various FIG documents and other information.

4.4 Reading & Book List

A number of magazines have features on FORTH, SOFT being one of them. This is a new magazine, and worthy of the support of FORTH users. FORTH Dimensions is the newsletter of FIG USA and is also very interesting. A number of articles have appeared in BYTE,

August 1980	A FORTH language 'special'
February 1981	Stacking Strings in FORTH
March 1981	A coding Sheet for FORTH

A growing number of books are now available, and this is by no means an exhaustive list.

Starting FORTH by Leo Brodie, published by Prentice-Hall (approx £16)

Threaded Interpretive Languages by R.G.Loeliger, published by McGraw-Hill (approx £16)

Invitation to FORTH by Harry Katzan Jr, published by Petrocelli (approx £12)

The Complete FORTH by Alan Winfield, published by Sigma Technical Press (£6.95 Our Recomended Book)

Using FORTH by Leo Brodie, published by FORTH Inc

All About FORTH by Glen Haydon, published by Mountain View Press (approx £20)

4.5 Multiple Copies

It is fundamental to the nature of FORTH, being an extensible language, that the language can be regenerated to incorporate any additions that you may care to add.

We are therefore quite happy to allow you to make copies of your FORTH software. We are also quite happy for you to make copies if you have more than one machine in your possession up to a limit of 5, but we would ask you not to make, or give copies to any third party (your friends, etc) or to sell any copies. If you have more than 5 machines in your possession, we would ask you to pay us a further 50% of your purchase price in order to use more machines. These comments regarding copies of the software also apply to the manual, and this is why we have chosen a very simple method of binding.

We hope you will agree with us when we say that it is only by Software Vendors, such as ourselves, making a reasonable return on our efforts, that the quality of the software market will improve and prosper. You must realise that if piracy is rife the best software will never be put onto the market and prices will remain high. We ask your co-operation in ensuring that this product is not abused in this way.

5.0 Compatibility

It is possible to make nas-FORTH compatible with FORTH-79 Standard definitions by compiling the following definitions.

```
CODE PICK
        H POP   H DCX   H DAD   SP DAD   M W' MOV   H INX
            M W MOV   W PUSH   NEXT JMP
        END-CODE


CODE ROLL
        EXX   H POP   H DAD   H I MOV   L I' MOV   H DCX
            H DCX   SP DAD   M W' MOV   H INX   M W MOV
            W PUSH   H W MOV   L W' MOV   H DCX   H DCX
            LDDR   EXX   H POP   NEXT JMP
        END-CODE


CODE J
        ' I 1+ @   H LXD   H INX   H INX   H INX   H INX
            M W' MOV   H INX   M W MOV   W PUSH   NEXT JMP
        END-CODE


CODE R@
        FORTH   ' R CFA @   ' R@ CFA !
        END-CODE

CODE EXIT
        FORTH   ' ;S CFA @   ' EXIT CFA !
        END-CODE


: D<        ( D1 D2 --- FLAG )
        DMINUS D+ SWAP DROP 0< ;


: MOVE      ( AD1 AD2 N --- )
        2 * CMOVE ;


: VARIABLE
        0 VARIABLE ;


: CREATE
        VARIABLE -2 ALLOT ;


: 0>        0 > ;
: 1-        1 - ;
: 2-        2 - ;
: >IN       IN ;
```

(continued)

HEX (continued)

```
: ?DUP     -DUP ;
: CONVERT        (NUMBER) ;
: DNEGATE        DMINUS ;
: NEGATE         MINUS ;
: NOT            0= ;
: SAVE-BUFFERS FLUSH ;
: U/MOD          U/ ;
: SIGN           0< IF 2D HOLD THEN ;
: 79-STANDARD ;
```

6.0 The Editor

When a BASIC program has been written and is working satisfactorily it may be saved as a named file. This is possible because a BASIC program is stored in source form - very similar, if not identical, to the way it is typed at the keyboard. Since FORTH is a compiled language it is not stored in source form, but as a list of addresses of routines. It is, therefore, to be expected that the writing of a FORTH application which is to be saved on tape or disc will be somewhat different from the corresponding method for BASIC.

In FORTH the source text is edited into a screen, whose contents can be compiled, tested and, if necessary, modified until its action is correct. The screen can then be saved and any further screens written in the same manner.

It is a good idea to develop all but the very simplest of applications by use of the editor since it allows the modification of a definition without the need for execessive retyping. The editing functions are placed in a separate EDITOR vocabulary. Note the difference between the commands:

```
FORTH VLIST    <enter>
```

and

```
EDITOR VLIST   <enter>
```

The best way to learn the actions of the editing facilities is, as with the rest of FORTH, by using them. The following is an example of how the EDITOR vocabulary can be used to write, modify and save an application.

Before the editor can be used a blank screen must be set up and the EDITOR vocabulary declared.

```
DECIMAL  EMPTY-BUFFERS    EDITOR      <enter>

1 CLEAR   <enter>         (Clear screen 1 to blanks)

L    <enter>              (Lists the current screen)

Screen 1
  0
  1
  2
  3
  4   etc.
```

We will now show an example of a program, and the first editor word used is P. We will explain this word later.

Type the following :

```
0 P ( RANDOM NUMBER GENERATOR )     <enter>
1 P DECIMAL     <enter>
2 P 0 VARIABLE SEED      <enter>
3 P : (RND)     <enter>
4 P       SEED @ 259 * 3 +      <enter>
5 P       32767 AND DUP SEED !  ;    <enter>
6 P : RND      ( RANGE -- RANDOM )  <enter>
7 P        (RND)   32767  */  ;-     <enter>
```

Typing L again will now give you the contents ....

```
Screen 1
  0 ( RANDOM NUMBER GENERATOR )
  1 DECIMAL
  2 0 VARIABLE SEED      etc, ...
```

In order to compile this source text, just as if it came from the keyboard, type 1 LOAD

We can now test some of our words, and change the if necessary.

```
100 RND .  81 ok
50 RND .  17 ok

: TEST CR 0 DO DUP RND . LOOP DROP ;     <enter>

10 10 TEST      <enter>
7 4 9 4 6 4 7 9 3 7  ok
```

In order to save this text to tape we must first make sure the buffer is put into the correct area by using FLUSH

Then we can save the whole screen area to tape using SCR-SAVE. You can apply a limit to SCR-SAVE by placing an address into the variable HI.

The EDITOR word P will put the following text onto a line of the screen. It expects to find the line number on the stack and is used as follows :

n P text for line n

It is conventional for line 0 of every screen to contain a comment giving the contents of that screen. A FORTH program can be rather difficult to follow, particularly if it makes great use of the stack so that variable names do not appear. For this reason it is a good idea to use plenty of comments when you write a screen. Since they are ignored when the screen is compiled they cost nothing in terms of dictionary space or execution time.

If a program extends to further screens the word --> should be put at the end of the screen. This is an instruction to continue interpretation with the next screen. The word --> is IMMEDIATE and so will execute even if the text is being compiled. This means that it is possible for a single definition to extend over a screen boundary. It is preferable not to do this as definitions should, if possible, be kept short. It is, however, sometimes necessary to write a definition (such as a machine code primitive) that cannot be fitted into a single screen, particularly as the screens are relatively short. A standard FORTH screen is 16 lines of 64 characters, which gives 1024 bytes or 1k.

If --> is not at the end of the screen, interpretation will stop at the end of the screen or at the word ;S. This word terminates the interpretation and may, therefore, be followed by further comments or instructions, which do not need to be enclosed in parentheses.

Editor Commands

P n ---
The next 64 characters from the input stream will be moved to line n of the current screen. Used in the form :
5 P THIS LINE WILL BECOME LINE 5 OF THE CURRENT SCREEN

H n ---
Hold line n at PAD (used more by the system than by the user)

D n ---
Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up one line.

R n ---
Replace line n with the text in PAD

I n ---
Insert the text from PAD at line n, moving the old line n and the following lines down. Line 15 is lost.

E n ---
Erase line n to blanks.

S n ---
Spread at line n. n and subsequent lines move down one line. Line 15 is lost.

L List the current screen.

DAVID HUSBAND

Producers of High Quality FORTH Software

Education – Industry – Business – Pleasure

Personal Callers By Appointment

VAT Reg. No. 355 8496 10

Telephone:
Bournemouth (0202) 764724

2 Gorleston Road
Branksome
Poole
BH12 1NW

AVAILABLE NOW!

"SPELL" – A FORTH Decompiler

How many times have you wanted to know how a word is defined ? "SPELL" will tell you. "SPELL" is the ultimate decompiler.

"SPELL" will :

* Decompile all High-level Definitions including :
  * Variables & Constants
  * <BUILDS and DOES>
  * All FORTH Structures
* Cope with headerless High-Level Code
* Tell you if the word is a code definition
* Show you all the header information including the NFA, LFA, CFA, and PFA
* Allow you to learn a great deal about FORTH
* Allow you to 'patch' your system

"SPELL" will not dis-assemble machine code – we are working on that at the moment! "SPELL" is written in FORTH and will work on any fig-FORTH or FORTH-79 system and is supplied as a four-page source listing which the user has to type into his machine.

"SPELL" is available now for £5 + VAT

The copyright of "SPELL" belongs to DAVID HUSBAND and we ask users not to reveal the source-code to others. DAVID HUSBAND has been using "SPELL" for the past three years, and we now feel that the market is ready for a superior decompiler, especially as there are now some simple decompilers around.

Overleaf are some examples of "SPELL" in action.

NOTE : SPELL WILL NOT WORK WITH 2X81-FORTH

E & O E

```
 ok
Copyright (c) 1983 DAVID HUSBAND
SPELL LIST

NFA 2435  84       LENGTH BYTE
    2436  4C        L
    2437  49        I
    2438  53        S
    2439  D4        T
    243A  DA 23    LINK TO (signon)
CFA 243C  6F 07    DOCOLON
PFA 243E  31 0E    DECIMAL
    2440  43 0C    CR
    2442  E1 05    DUP
    2444  91 09    SCR
    2446  14 07    !
    2448  2E 0F    (.")
    244A  06       LENGTH-BYTE OF STRING
    244B  42        B
    244C  6C        l
    244D  6F        o
    244E  63        c
    244F  6B        k
    2450  20
    2451  D2 22    .
    2453  6B 01    LIT
    2455  10 00    $10
    2457  FE 07    0
    2459  2C 02    (DO)
    245B  43 0C    CR
    245D  48 02    I
    245F  1F 08    3
    2461  C2 22    .R
    2463  61 0C    SPACE
    2465  48 02    I
    2467  91 09    SCR
    2469  DD 06    @
    246B  AA 18    .LINE
    246D  E2 01    (LOOP)
    246F  EC FF     $245B
    2471  43 0C    CR
    2473  94 04    ;S

ok
Copyright (c) 1983 DAVID HUSBAND
SPELL CR

NFA C3E   82       LENGTH BYTE
    C3F   43        C
    C40   D2        R
    C41   EA 0B    LINK TO ^EMIT
CFA C43   6F 07    DOCOLON
PFA C45   6B 01    LIT
    C47   0D 00    $D
    C49   DE 0B    EMIT
    C4B   6B 01    LIT
    C4D   0A 00    $A
    C4F   DE 0B    EMIT
    C51   FE 07    0
    C53   88 09    OUT
    C55   14 07    !
    C57   94 04    ;S

ok
Copyright (c) 1983 DAVID HUSBAND
```

```
234 LIST

 0 IN THIS PAPER WE WILL DEMONSTRATE THAT
 1 ON THE ONE HAND, STUDIES HAVE SHOWN THAT
 2 ON THE OTHER HAND, HOWEVER, PRACTICAL EXPERIENCE INDICATES THAT
 3 IN SUMMARY, THEN, WE PROPOSE THAT
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
235 LIST

 0 BY USING
 1 BY APPLYING AVAILABLE RESOURCES TOWARDS
 2 WITH STRUCTURED DEPLOYMENT OF
 3
 4 COORDINATED WITH
 5 TO OFFSET
 6 BALANCED BY
 7
 8 IT IS POSSIBLE FOR EVEN THE MOST
 9 IT BECOMES NOT UNFEASABLE FOR ALL BUT THE LEAST
10 IT IS NECESSARY FOR ALL
11
12 TO FUNCTION AS
13 TO GENERATE A HIGH LEVEL OF
14 TO AVOID
15
```

```
236 LIST

 0 INTEGRATED          MANAGEMENT          CRITERIA
 1 TOTAL               ORGANIZATIONAL      FLEXIBILITY
 2 SYSTEMATIZED        MONITORED           CAPABILITY
 3 PARALLEL            RECIPROCAL          MOBILITY
 4 FUNCTIONAL          DIGITAL             PROGRAMMING
 5 RESPONSIVE          LOGISTICAL          CONCEPTS
 6 OPTIMAL             TRANSITIONAL        TIME PHASING
 7 SYNCHRONIZED        INCREMENTAL         PROJECTIONS
 8 COMPATIBLE          THIRD GENERATION    HARDWARE
 9 QUALIFIED           POLICY              THROUGH-PUT
10 PARTIAL             DECISION            ENGINEERING
11 STAND-ALONE         UNDOCUMENTED        OUTFLOW
12 RANDOM              CONTEXT SENSITIVE   SUPERSTRUCTURES
13 REPRESENTATIVE      FAIL-SAFE           INTERACTION
14 OPTIONAL            OMNIRANGE           CONGRUENCE
15 TRANSIENT           UNILATERAL          UTILITIES
```



```
237 LIST

 0 ( BUZZPHRASE GENERATOR II -- MARGIN SETTING)      EMPTY
 1 181 LOAD (~RANDOM NUMBERS)
 2 32 CONSTANT BL                78 CONSTANT RMARGIN
 3 VARIABLE LINECOUNT            VARIABLE HOMEBASE  2 ALLOT
 4 : <WRITE   BLK @  >IN @  HOMEBASE 2! ;
 5 : WRITE>   HOMEBASE 2@  >IN !  BLK ! ;
 6
 7 : CR   CR  0 LINECOUNT ! ;
 8 : SPACE   LINECOUNT @  IF SPACE  1 LINECOUNT +!  THEN ;
 9 : .WORD   ( adr)  COUNT DUP LINECOUNT @ + RMARGIN >
10          IF CR ELSE SPACE THEN
11          DUP  LINECOUNT +!  TYPE ;
12 : ANOTHER  ( lim -- lim adr)  BL WORD  OVER >IN @ < NOT ;
13 : WORDS   ( u)
14         >IN @ +  BEGIN ANOTHER WHILE .WORD REPEAT  2DROP ;
15 238 LOAD    239 LOAD
```

```
238 LIST

 0 ( BUZZPHRASE GENERATOR -- HIGH LEVEL WORDS)
 1
 2 : BUZZ   16 CHOOSE 64 * + >IN !  236 BLK !  20 WORDS ;
 3 : 1ADJ   0 BUZZ ;
 4 : 2ADJ   20 BUZZ ;
 5 : NOUN   40 BUZZ ;
 6 : PHRASE   1ADJ 2ADJ NOUN ;
 7 : FILLER   ( u)  [ 4 64 * ] LITERAL *
 8       3 CHOOSE 64 * + >IN !  235 BLK !    64 WORDS ;
 9 : SENTENCE   4 0 DO I FILLER PHRASE LOOP ." ." CR ;
10 : INTRO   ( u)   64 * >IN !  234 BLK !  CR 64 WORDS ;
11
12 : PAPER   <WRITE  CR CR  4 0 DO I INTRO SENTENCE LOOP WRITE> ;
13
14
15 : TEST   CR '  <WRITE EXECUTE WRITE> SPACE ;
```

```
239 LIST

 0 ( RETRIEVAL OF MORE SUCCESSFUL PAPERS)
 1
 2 VARIABLE SEED
 3
 4 : 4POSTERITY   RND @ SEED ! ;
 5 ( execute BEFORE producing a paper)
 6
 7 : REDO   SEED @ RND ! ;
 8 ( execute AFTER a paper, to reprint it.
 9   Usage:  REDO PAPER   )
10
11
12
13
14
15
```