C64

Melbourne House

COMMODORE 64 FORTH+

John Jones-Steele

Chapter 1

INTRODUCTION

STARTING OUT

This manual is not intended to be a tutorial of FORTH as there are many books available on the market that do this more than adequately. However, Chapter 2 gives a short but useful guide to the basics of FORTH, along with the extensions presented for the Commodore 64. For the beginner to the language, a good book is 'Starting Forth' by Leo Brodie, even though this describes FORTH-79 rather than fig-FORTH. For the more advanced user, a useful reference is 'The Systems Guide to fig-FORTH' by C. H. Ting.

FORTH is a language which combines the features of high-level languages with the speed of machine code. The FORTH interpreter interprets each line of commands by searching through an internal dictionary of words which it understands. It then acts on these words. Some of the words allow creation of new words, which in turn form the basis for even higher-level words. Eventually a whole program comes together, and is called by a single word. This gradual development allows for better checking of sections of a program than BASIC does.

SETTING UP

To install the basic system on your Commodore 64, type LOAD, then press RETURN. Commodore 64 FORTH+ uses PAVLODA and will load in less than 2 minutes.

When the compiler has loaded, you will be greeted by the message:

```
COMMODORE 64 FIG-FORTH+ 1.1A
(C) ABERSOFT:1984
```

and a triangle as a cursor.

If you have a disc system, you can convert your new FORTH system to run on disc, by typing:-

```
DISC
```

followed by

```
HERE ENV
```

This creates a file on disc called @MCODE. This can be renamed to anything you like, but a sensible name would be FORTH! To use this as your default system, use:-

```
LOAD "FORTH",8,1
SYS 16384
```

All commands that would have used tape will now use disc. If you need to go back to the tape system,

```
TAPE
```

will now reference tape only

RUN/STOP can be trapped by ?TERMINAL, but if you get into a loop, RUN/STOP RESTORE will do a WARM start. as long as the dictionary has not been corrupted.

To test that the system is running, just hit the RETURN key and the system should respond

```
OK
```

You are now ready to begin programming in FORTH.

Chapter 2

BEGINNERS' GUIDE TO FORTH

THE STACK

One of the major differences between FORTH and most other high-level languages is that FORTH uses Reverse Polish Notation (RPN). Normally, the operator '+' comes between the numbers you wish to add (e.g. 4 + 7). In RPN, the operator comes after the numbers (i.e. 4 7 +) instead. Note that you need to insert spaces between the 4, 7 and +. This is because a stack is used to store numbers when evaluating expressions. (In case you do not know what a stack is, imagine a pile of plates. The last plate put on the pile, being on top, is the first to come off again.) When FORTH finds a number, it puts it onto the stack. When it finds an operator, it takes the required numbers off the stack, then puts the result back onto the stack. The word dot '.' (without the quotes) takes a number off the top of the stack and prints it.

The BASIC program line

```
PRINT (3+7)*(8+2)
```

is, in FORTH,

```
3 7 + 8 2 + * .
```

(note: the spaces are important)

After executing:	the stack becomes:
3	(3) top of stack on this side
7	(3,7)
+	(10)
8	(10,8)
2	(10,8,2)
+	(10,10)
*	(100)
.	() 100 is printed.

Practice is the only way to get familiar with RPN. Probably the best thing to do now, if you are a beginner to programming, is to sit at your keyboard and try a few examples. After a short while, RPN will become as easy as normal arithmetic and you will find no problem in your application programming. You may find sometimes that you get the message ?MSG 1 when you try to do something. This merely means that you tried to use more information than you have placed on the stack.

Similar words explained in the reference guide are:

```
+ – / */ */MOD /MOD MOD MAX MIN AND OR XOR MINUS +– 1+ 2–
```

There are some words which shuffle the numbers on the stack:

DUP	duplicates the top number
DROP	discards the top number
SWAP	swaps the two top numbers
ROT	rotates the top three numbers, bringing the third to the top.
OVER	carries the second value over the top as the new first value.

So, for example:

```
1 DUP . .   prints 1 1 OK
1 2 DROP .  prints 1 OK
1 2 SWAP . .  prints 1 2 OK
```

See also S0 SP@

VARIABLES AND CONSTANTS

A variable in FORTH must be explicitly created before it is used, using the word 'VARIABLE'.

```
3 VARIABLE A1
```

will create a variable 'A1' with initial value 3. Any sequence of alpha-numeric characters will work as a name.

```
6 A1 !
```

stores 6 in A1 once A1 has been defined as a variable. (This is equivalent to A1=6 in BASIC.)

```
A1 @ .      (can also be written as A1 ?)
```

will fetch the contents of 'A1' and print it. The word 'A1' actually puts the address of 'A1' on the stack. '!' takes an address off the stack, then takes a number off the stack and stores it at that address. The word '@' takes an address off the stack and replaces it with the contents of memory at that address. The word '?' is the equivalent of '@ .' for those who are lazy. See also the word '+!'.

A constant in FORTH is a fixed number which is given a name, using the word, 'CONSTANT'.

```
10 CONSTANT TEN
```

creates a constant, 'TEN', with the value 10. From now on, each time the word 'TEN' is found, 10 will be pushed onto the stack, so

```
TEN .
```

will print:

```
10 OK
```

Note that '!' and '@' are not needed with constants, and in fact ! cannot be used to change a CONSTANT, because CONTANTS are constant!

Among the predefined system variables there is 'BASE'. 'BASE' contains the current number base used for input and output. 'HEX' stores 16 (base 10) in 'BASE', setting hexadecimal mode. 'DECIMAL' stores 10 (base 10) in 'BASE', setting decimal mode.

TYPES OF NUMBERS

FORTH is a typeless language. There is no distinction between a number and a character, for example. Things on the stack are taken by a word and interpreted as that word sees fit. Some of the more common interpretations, other than the normal 16 bit signed numbers (range –32768 to 32767) which have been used, are:

Unsigned

Range 0 to 65535. The word 'U.' acts like '.' but it assumes that the number is unsigned rather than signed.

Double precision

The top two numbers on the stack are interpreted as a 32 bit number, either signed or unsigned. The top number is taken as the high 16 bits; the second number is taken as the low 16 bits. To put a 32 bit number onto the stack, type the number with a decimal point somewhere in the number. The system variable DPL contains the number of digits following the decimal point, in case it is important.

```
70.000 . . DPL @ .
```

will print 1 4464 3 OK. The 32 bit number 70000 (not 70) is put onto the stack as two numbers. The first '.' prints the top half as a signed number. The bottom half is then printed (note: 70000 = 1 × 2 ↑ 16 + 4464). The contents of DPL are 3, since there are 3 digits after the decimal point.

The words U* U/MOD M* M/ and M/MOD are mixed mode versions of * / and /MOD where the operands and results are of different types.

Byte

Sometimes only 8 bit numbers are required, for instance to represent a character. They are represented on the stack by 16 bit numbers with the 8 highest bits zero. 'C@' and 'C!' fetch and store only a single byte at a time.

Flags

To make decisions, the values true and false are needed. For instance:

```
2 3 = .
```

prints 0 (false), whereas:

```
3 3 = .
```

prints 1 (true). In fact, any non-zero number is treated as true. (So, '–' can be used for <>, since x–y=0 (false) only if x=y.) Other comparisons are:

```
< U< 0<  (equivalent to 0 <)  and 0=  (equivalent to 0 =).
```

'NOT' changes a true flag to false and vice versa. (This is actually equivalent to 0=.)

FORTH MODES

FORTH has two different ways of working. One is the interpretive mode and the other is the defining (or compiling) mode.

In the interpretive mode, FORTH takes whatever is input and tries to execute it immediately. If you type

```
4 7 + .
```

the computer will give an immediate answer of

```
11 OK
```

However in compiling mode, FORTH takes what is input and stores it away in its dictionary and uses it later. As an example, if you wanted to input the sum above but only see the result later (a rather strange thing to want to do), you could define a new word, thus:

```
: STRANGE 4 7 + . ;
```

(: is the word used by FORTH to mean begin compiling and ; is the word for finishing compiling). If you then type

```
STRANGE
```

and 'enter', the computer will then give you

```
11 OK
```

Once a word is defined in this way, it can be used in other definitions:

```
: TOOSTRANGE STRANGE STRANGE ; TOOSTRANGE
```

compiles a word 'TOOSTRANGE' which will perform STRANGE twice, then executes the word, printing

```
11 11 OK
```

Note that as far as FORTH is concerned, once a word has been defined using : … ; it then becomes part of the language and can be used in exactly the same way as any other word, like

SWAP and DROP. When defining new words, try to name them so that they mean what they do. In the middle of a large piece of work, a definition named CENT-FAHR is pretty obvious, whereas CF will not be.

As an example of a more useful FORTH definition than STRANGE, here is CENT-FAHR:

```
: CENT-FAHR          ( expects a single number on the stack)
        9 *          ( multiply by 9)
        5 /          ( divide by 5)
        32 +         ( add 32)
        .            ( print the answer)
;                    ( definition finished)
```

Now, typing 10 CENT-FAHR (return) will give a screen display of

```
10 CENT-FAHR 50 OK
```

CONTROL STRUCTURES

The FORTH structure IF … ENDIF (or IF … ELSE … ENDIF) allows conditional execution (only) within a definition. The equivalent of BASIC's 'IF A>2 THEN PRINT "TOO BIG" ' is

```
: TEST A @ 2 > IF ." TOO BIG" ENDIF ;
```

'A @ 2 >' puts a flag on the stack stating whether A is greater than 2. IF removes the flag. If the flag is true (non-zero), then the following code is executed. The word '."' prints everything up to the next double quote '"'. (Note that there has to be a space after '."' for it to be recognised properly.) If the flag is false (zero), then the code up to ENDIF is skipped. In the case of IF … ELSE … ENDIF, if the flag is true, the code between IF and ELSE is executed and the code between ELSE and ENDIF is skipped. If the flag is false, only the ELSE … ENDIF code (and that which follows ENDIF as usual) is executed. Inside any IF … ENDIF clause you can have another one.

BASIC's 'FOR A = 1 TO 10 : PRINT A : NEXT A' is FORTH's

```
: TEXT 11 1 DO I . LOOP ;
```

When 'TEXT' is executed, 'DO' takes two numbers off the stack. The top number (1 here) is the starting value. The second number (11) is the limit. 'I' copies the loop index (A in the BASIC program) onto the stack, where it is printed by '.'. 'LOOP' increments the loop index by 1. If the limit is **reached** or exceeded, execution continues as normal, otherwise it loops back to the 'DO'. Hence the limit being 11 in order to count to 10. Note that the loop is done at least once, no matter what the limit is. 'n +LOOP' instead of 'LOOP' is FORTH's equivalent of 'STEP n' in BASIC. 'LEAVE' changes the limit so that the loop will be left as soon as 'LOOP' is reached. 'J' returns the value of the outer loop counter if you have a nested loop structure. For example:

```
: DEMO CR
        4 1 DO
            9 7 DO
                I J . . CR
            LOOP
        LOOP
;
```

will print:

```
1 7
1 8
2 7
2 8
3 7
3 8
```

Another type of loop is BEGIN … flag UNTIL.

```
: TEST BEGIN … A @ 2 > UNTIL ;
```

is the equivalent of

```
10 REM BEGIN
20 …
30 IF NOT(A>2) THEN GOTO 10
```

i.e. … will be executed over and over until A>2.

This is a very useful instruction when we need to repeat a loop over and over until the RUN/STOP key is pressed. This can be done like so:

```
: TEST BEGIN ." LOOPING" CR ?TERMINAL UNTIL ;
```

The ?TERMINAL leaves a flag on the stack of true if RUN/STOP is pressed and false if it is not.

If a loop is required to continue forever, BEGIN … AGAIN can be used. This has no conditions and would be very useful when writing a game program, e.g.

```
: GAME
        TITLES
        BEGIN
                PLAY-GAME
                END-MESSAGE
        AGAIN
;
```

This also shows how programs should be put together, with the final word used to call a program, consisting of only a few, already tested and built up words.

The problem with the above conditional structures is that they are all executed at least once, as the test is carried out at the bottom of the loop. Certain programs will require the test to be carried out at the top of the loop. This is done with the BEGIN … flag WHILE … REPEAT structure. For example:

```
: DEMO
        BEGIN DUP 4 > WHILE
        DUP . 1 – REPEAT
        DROP
;
```

If 7 DEMO is entered, the output will be 7 6 5; if 3 DEMO is entered, there will be no output.

One of the most useful commands in a language such as Pascal is the CASE statement. This allows for the testing of a number for many different values and executing different procedures on each value. This is available in standard FORTH only by using many nested IF … IF … IF … ENDIF ENDIF ENDIF and can be very tedious. A new structure in Abersoft FORTH is naturally called the CASE structure. Its use is:

```
… (instructions leaving a single value on the stack)
CASE
        8 OF ." This is 8" CR ENDOF
        12 OF ." This is 12" CR ENDOF
        99 OF ." This is 99" CR ENDOF
ENDCASE
```

This structure provides a much more readable and less error-prone way of making multiple decisions. In the above example, if 99 was left on the stack: This is 99 : would be printed. Any value other than 8, 12 or 99 would produce no output at all.

KEYBOARD AND SCREEN I/O

You have met the word ' ." ' which prints a fixed message on the screen. 'EMIT' expects the ASCII code for a character on the stack. It then prints that character.

```
65 EMIT
```

will print the letter 'A' (ASCII code 65). 'TYPE' is used for printing strings. The variable 'TIB' contains the address of the Terminal Input Buffer where what you type is stored. To type out the first 10 characters in the buffer, you need to supply 'TYPE' with the starting address, and the number of characters to be printed:

```
TIB @ 10 TYPE
```

will print

```
TIB @ 10 T
```

'SPACES' takes a number n from the stack and prints out n spaces.

'KEY' waits for a key to be pressed, then puts the ASCII code of that key onto the stack.

'EXPECT' requires an address and a count, just like 'TYPE'. However, 'EXPECT' takes the specified number of characters from the keyboard (or all the characters up to 'enter') and tacks one or two nulls (ASCII 0) on the end. The word 'QUERY' is defined as 'TIB @ 80 EXPECT'.

'GET' is similar to 'KEY' but whereas 'KEY' only allows ASCII codes 20,13 and 32 to 127 to be input and switches on the cursor, 'GET' returns all key values, without the cursor appearing.

VOCABULARY

'VLIST' prints out all known words in the current vocabulary. (A vocabulary is a subsection of the whole dictionary. The normal vocabulary is called FORTH and is selected using the word 'FORTH'.)

FORGET word

will forget all words defined from 'word' onwards. A handy thing to do is to compile the word 'TASK' as the first new word:

```
: TASK ;
```

Then, if after compiling more words, you decide to get rid of them all,

```
FORGET TASK
```

will do the job. The system variable 'FENCE' contains an address, below which you cannot FORGET a word. To protect the words you have just compiled, type 'HERE FENCE !'

To create a new vocabulary 'MYWORDS' type

```
VOCABULARY MYWORDS IMMEDIATE
```

The word 'MYWORDS' will now cause this new vocabulary to be searched when interpreting words (the system variable 'CONTEXT' is set to MYWORDS). New definitions will, however, be added to the old vocabulary (the system variable 'CURRENT' is still pointing to FORTH). To select the new vocabulary as the one to add new definitions to, type:

```
MYWORDS DEFINITIONS
```

This sets the current vocabulary to the context vocabulary (made MYWORDS by 'MYWORDS'). To go back to adding definitions to the FORTH vocabulary, type

```
FORTH DEFINITIONS
```

Some words, such as 'FORTH', are immediate. This means that they will be executed, even in compile mode.

THE RAM DISC

One of the problems with using FORTH on a non-disc system is that once a word has been defined, the original source for that definition is lost. For a large definition, this is an obvious nuisance if it is found that it does not subsequently do what was expected of it. In Abersoft FORTH (cassette version), this has been circumvented by using the top of the store as a small, pretend disc of 11k bytes in total size. This may seem like a small amount but given FORTH's compactness, a surprisingly large application can be written. An example, described later, took up only 4 pages of this disc (a page is 960 bytes long arranged as 24 lines of 40 characters).

The pages on the disc are numbered from 0 to 10, page 0 being reserved for comments; text is then input to the disc by means of the editor described later. To compile a program from RAM-disc, use

n LOAD (where n is the page number of the first definition)

If the definition, or definitions, spread over more than one page, the final word on the page should be

--> (pronounced next-screen)

To prepare the RAM disc for writing, the command

INIT-DISC

should be used. This simply clears the area with blanks. To list the current contents of the disc pages, use

n LIST (where n is the page to be listed)

When a disc has been filled, it can be saved to tape or disc with the command

n SAVER (where n is a number from 1 to 26)

To reload a disc area created previously, use

n LOADR (where n is a number from 1 to 26)

but remember, this overwrites whatever is already there.

THE GRAPHIC ROUTINES

These routines are the major extensions to the FORTH standard, and allow use of all the Commodore 64's hi-res graphics capabilities, which, with the speed of Abersoft FORTH, allow fast-action, arcade-type games to be written without the need to resort to machine code.

n1 n2 AT

(where n1 = line number and n2 = column)

AT positions the cursor at n1, n2.

n INK
n BORDER
n BACKn1

(where n equals a Commodore colour value from 0-15)

INK sets the character ink colour; BORDER sets the border; and BACKn1 sets background colour register n1.

MCM
NCM
EBCM
NBCM
HRG
LRG

MCM sets the multi-colour mode; NCM resets normal colour mode; EBCM sets extended background colour mode; NBCM sets normal background colour mode; HRG sets hi-res graphics; and LRG sets lo-res.

CLG
n CLB

CLG clears the hi-res screen; CLB clears the hi-res background to colour n.

n1 n2 PLOT

PLOT plots the point n1, n2 on the Hi-Res screen.

UDGS
ROMC
COPYC

UDGS sets the character usage to the UDG area; ROMC resets to the standard ROM character set; and COPYC copies the ROM set to the UDG area, for further editing.

SPRITE COMMANDS

n1 n2 n3 n4 n5 DEFSPR
n1 n2 SETSPR

There is room for 16 physical sprites in the fig-FORTH+ system. More can be defined by moving memory around. DEFSPR defines line n4 of physical sprite n5 as n1, n2 and n3. SETSPR sets logical sprite n2 to physical sprite n1.

n SPMON
n SPMOFF
n SPRON
n SPROFF
n HEXPON
n HEXPOFF
n VEXPON
n VEXPOFF

SPRON & SPROFF set logical sprite n on or off; SPMON & SPMOFF set logical sprite n to multi-colour mode; HEXPON, HEXPOFF, VEXPON & VEXPOFF set the horizontal and vertical expansion on logical sprite n.

n1 n2 SPCOL
n SPMCOL0
n SPMCOL1

SPCOL sets logical sprite n2 to colour n1; SPMCOL0 & 1 sets the sprite multi-colour registers to colour n.

n1 n2 n3 SPRITE

SPRITE moves logical sprite n3 to n1, n2 (x,y).

n PRION
n PRIOFF

PRION sets logical sprite n priority over data; PRIOFF sets data priority over logical sprite n.

SSCOL n
SDCOL n

SSCOL returns a byte holding the bits of sprite to sprite collision; SDCOL returns a byte showing which sprites have hit data. Both these commands clear the register that holds the data, so if multiple testing is required duplicate the byte on the stack, or save it in a variable.

SOUND ROUTINES

CLRSND

CLRSND clears all sound registers and sets them up for use.

n1 n2 FREQ
n1 n2 PULSE
n1 n2 WAVE
n1 n2 ATTACK
n1 n2 DECAY
n1 n2 SUSTAIN
n1 n2 RELEASE

All these commands set the relevant part of voice n2 to n1.

n CUTOFF
n RESONANCE
n FILTER
n VOLUME
n FMODE

These commands set the relevant command register to n.

n VS3

This command sets voice 3 on or off depending on the value of n.

OS3 n
ENV3 n

OS3 returns the value of oscillator 3 in n; ENV3 returns the value of envelope 3 in n.

MISCELLANEOUS ADDITIONS

The final changes to the standard are:

FREE

This returns a value on the stack of the amount of store remaining in bytes. For example:

```
: BYTES FREE . .“ bytes remaining” CR ;
```

When BYTES is typed, the message ‘11919 bytes remaining’ or similar will be printed.

SIZE

This returns a value on the stack of the current size of the dictionary.

n1 PADDLE n2
n1 JOYSTICK n2

These commands return the value of JOYSTICK or PADDLE n1 in n2.

Chapter 3

ADVANCED FEATURES OF FORTH

Memory Map

0000-00FF	Zero Page
0100-01FF	Reserved SP
0200-03FF	Op System
0400-07FF	Normal Screen
0800-0BFF	HRG Colour Matrix
0C00-0FFF	Physical Sprite Data (0-15)
1000-1FFF	Character Set/Workspace
2000-3FFF	HRG Screen (or User Defined Graphics)
4000-9FFF	Forth Dictionary
A000-CC00	RAM-Disk

SAVING AN EXTENDED DICTIONARY

As can be seen from any work on FORTH, a completed FORTH application is simply an extension of the dictionary. In Abersoft FORTH, if you have a completed program and do not wish to compile from RAM-disc each time you want to use it, or you have a set of standard routines that you wish to use every time you enter FORTH, these may be saved by the following method:

1 Change the COLD START parameters by typing the following:

```
FORTH DEFINITIONS DECIMAL
LATEST 12 +ORIGIN !
HERE 28 +ORIGIN !
HERE 30 +ORIGIN !
HERE FENCE !
' FORTH 8 + 32 +ORIGIN !
```

(If using a vocabulary different from FORTH, use ' vocab 6 + …)

2 Type:

```
HERE ENV
```

to save your new extended version of FORTH to tape or disc. (Use your own tape to do this on, *NOT* the master FORTH tape.) This new tape is, of course, for your own use only and not for re-sale, hire or lending purposes.

3 When you have a stand alone program, that will never need to be changed, or if you want to create a program for re-sale, you need to make sure that the application will never return to the command level, so type:

```
ZAP word
```

where word is the final word of the program, like the word GAME in the example in Chapter 2. This will then save an image of the system to tape or disc. This new program can be run with SYS 16384.

REGISTER USAGE

The programmer who wishes to use machine code routines (using CREATE or ;CODE) or Assembler, will require the register usage of Abersoft FORTH. These are as follows:

FORTH REGISTERS

IP	address of the Interpretive Pointer
W	address of the pointer to the code-field
UP	User Pointer containing the address of the base of the user area
N	a utility area in zero page from N–1 to N+7

6502 REGISTERS

When FORTH leaves NEXT to execute CODE, the following conventions apply:

1. The Y-register is zero and can be freely used.
2. The X-register defines the low byte of the data stack relative to address 0000.
3. The 6502 stack-pointer points one byte below the low byte of the bottom of the return stack pointer.
4. The accumulator may be freely used
5. The processor is in binary mode and must be returned that way.

To understand this area fully will require Ting's book, as mentioned earlier.

Chapter 4

FORTH ASSEMBLER

ASSEMBLER

This allows easy construction of either full words using:

CODE … (assembly mnemonics) … END-CODE

or new defining words using:

: … ;CODE … (assembly mnemonics) … END-CODE

For example:

```
CODE DOUBLE          ( take top word of stack and double it)
CLC,                 ( clear the carry bit)
BOT ROL,             ( rotate left the 8 byte value referenced by 0,X)
BOT 1+ ROL,          ( and the next one)
NEXT JMP,
END-CODE
```

As can be seen from the above example, FORTH Assemblers are structured in Reverse Polish, as is the rest of the language. The complete FORTH Assembler is explained below.

All op-codes are as in standard 6502 code, except for the addition of a trailing comma, e.g. STA, CLC, etc. When terminating a FORTH word, control must return to either NEXT, which executes the next FORTH word, or to one of the following:

POP	Remove one 16 bit value from the data stack.
POPTWO	Remove two 16 bit values from the data stack.
PUSH	Add two bytes to the data stack.
PUT	Replace the bottom two bytes by overwriting.

The convention when using PUT and PUSH is for the accumulator to hold the high byte, and the stack the low byte.

The different addressing modes in 6502 are addressed as follows:

SYMBOL	**MODE**	**OPERAND**
.A	accumulator	none
#	immediate	8 bits only
,X	indexed X	z-page or absolute
,Y	indexed Y	z-page or absolute
X)	indexed indirect X	z-page only
)Y	indirect indexed Y	z-page only
)	indirect	absolute only
none	memory	z-page or absolute

For example,

```
1 # LDY,             ( LDY #1)
DATA ,X STA,         ( STA DATA,X)
6 X) ADC,            ( ADC (6,X)
```

The bottom of the stack and the next to bottom are referenced often enough that the words BOT and SEC are included.

```
BOT LDA,        assembles LDA (0,X)
SEC LDA,        assembles LDA (2,X)
```

The return stack resides in the 6502 stack area (page 1) and is referenced by the X-register. Therefore, if you need the X-register in your definitions, it must be preserved. A word is defined for you to save the X register in, thus:

```
XSAVE STX,
```

When your word is finished, before returning to the interpreter,

```
XSAVE LDX,
```

This must be done, else a CRASH will result.

As was mentioned in the previous chapter, there is a data area available in zero-page, accessed by the FORTH word N. To assist in your use of this area, a machine code subroutine is provided. This moves data from the data stack into the N area. Either one, two, three or four 16 bit values may be moved. For example, to move the bottom two 16 bit words from the stack into the N area, use:

```
2 # LDA,   SETUP JSR,
```

Remember that this actually removes values from the stack, not just copies them.

Jumps and loops can be used as in normal assembly language, but this can be difficult. FORTH Assembler allows similar looping structures to normal FORTH. These are

```
BEGIN, … cc UNTIL,
```

This is the equivalent of

```
L1:     mnemonics
        Bcc L1
```

```
cc IF, … ENDIF,
cc IF, … ELSE, … ENDIF,
```

(where cc equals

```
CS          test carry set
0<          byte less than zero
0=          equal to zero
```

and the above followed by NOT to indicate the reverse condition.)

The FORTH assembler does not have many checks on the way you write your code, and can easily allow illegal instructions to be used Checks are made at the end of assembly that the stack is clear, but the onus is on YOU for most error checking.

Chapter 5

THE DISC EDITOR

ARRANGEMENT OF THE RAM-DISC

FORTH organises all mass storage as screens of 960 characters (actually 1024 for fig compatibility). The RAM-disc has 11k and its screens are numbered 0 to 11.

Each screen is organised by the system into 24 lines of 40 characters per line.

INPUTTING TO A SCREEN

To start an editing session, type SCREDIT. This invokes the SCREEN EDITOR vocabulary and allows text to be input.

The screen to be edited is selected, using

```
n EDIT   (list screen n and select for editing)
```

To clear a screen before use

```
n CLEAR   (clear screen n)
```

When using a RAM-disc for the first time, INIT-DISC should be typed to initialise the RAM to all blanks, otherwise the Commodore screen commands will probably make a mess of the screen layouts.

The editor starts in OVERWRITE mode. The cursor can be moved by using the arrow keys, and anything typed will be entered on the screen. RETURN just moves the cursor to the beginning of the next line. DEL deletes the previous character, as would be expected. Care should be taken at the end of the screen lines, as FORTH treats the screen as one continuous block, so if a word finishes in column 40 it will take column 1 of the next line as part of the previous line. The other mode is INSERT mode: toggling between the modes is accomplished with INST. In this mode, all characters entered move characters along the screen line you are on, losing excess off the right-hand side. DEL closes up a line, and RETURN performs the same function as in OVERWRITE mode. Two other commands, CONTROL S and CONTROL D, either open up a gap in the screen at the current cursor position, losing the 24th line, or delete the current line, filling the 24th line with spaces. To complete the edit, STOP is pressed. The option is then given to either complete the edit, or quit the edit leaving the screen as it was.

SAMPLE PROGRAM

A sample C64 Forth+ program is included on the cassette tape.

To load this program load FORTH, then enter 5 LOADR (this loads 5 pages of Forth source code into the RAM disk).

After loading is complete enter 1 LOAD (this Loads the RAM disk into FORTH) and then enter the word GAME (this tells FORTH to compile and run the game).

You can of course look at and change the program as you wish in order to gain familiarity with C64 FORTH+.

Chapter 6

ERROR MESSAGES

If the compiler finds an error at any point, it clears both the data and return stack, and gives an error message with a numeric value. The meaning of these error messages is:

MSG#	MEANING
0	Word not found
1	Stack empty
2	Dictionary full
3	Has incorrect address mode
4	Is not unique
6	RAM Disc Range? (not pages 0 to 10)
7	Full Stack
9	Trying to load from page 0
17	Compilation only use in a definition
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

Chapter 7

FORTH REFERENCE GUIDE

The reference guide contains all of the word definitions in this release of fig-FORTH (the extensions for the Commodore 64 have been presented in earlier chapters) in the main vocabulary. 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 procedure 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
d	32 bit signed double integer
f	boolean flag. 0=false, non-zero=true
ff	boolean false flag = 0
n	16 bit signed integer number
u	16 bit unsigned integer number
tf	boolean true flag = non-zero

Unless otherwise noted, all references to numbers are for 16 bit signed integers. The high byte is on top of the stack, with the sign in the leftmost bit. For 32 bit numbers, the most significant part is on top.

All arithmetic is implicitly 16 bit signed integer, with error and under-flow indication unspecified.

Acknowledgements are duly made to the FORTH INTEREST GROUP for parts of this compiler and manual, and it is recommended that membership of this august body is taken out The address of FIG(UK) is:

The Membership Secretary
FIG(UK)
24 Western Avenue
Woodley
Reading
RG5 3BH

! n addr ---

Stores 16 bits of n at address. Pronounced "store".

!CSP

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

d1 --- d2

Generates 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.

#BUF --- n

A constant returning the number of disc buffers allocated.

#S d1 --- d2

Generates ASCII text in the text output buffer, by the use of #, until a zero double number results. Used between <# and #>.

' --- addr

Used in the form:

' nnnn

it leaves the parameter field address of dictionary word nnnn. As a compiler directive, executes in colon-definition to compile the address as a literal. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Pronounced "tick".

(

Used in the form:

(cccc)

it ignores 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.

(.")

The run-time procedure compiled by ." which transmits the following in-line text to the selected output device. See ."

(;CODE)

The run-time procedure, 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 ---

The run-time procedure 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 procedure.

(DO)

The run-time procedure 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 to 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

Converts the line number n1 and the screen n2 to the disc buffer address containing the data. A count of 40 indicates the full line text length.

(LOOP)

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

(NUMBER) d1 addr1 --- d2 addr2

Converts 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 unconvertible digit. Used by NUMBER.

***** n1 n2 --- prod

Leaves the signed product of two signed numbers.

***/** n1 n2 n3 --- n4

Leaves 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

Leaves the quotient n5 and remainder n4 of the operation n1*n2/n3. A 31 bit intermediate product is used as for */.

+ n1 n2 --- sum

Leaves the sum of n1+n2.

+! n addr

Adds n to the value at the address. Pronounced “plus-store”.

+– n1 n2 --- n3

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

+BUF addr1 --- addr2 f

Advances 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)

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 of (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

Leaves 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.

.CPU

Prints the message ‘COMMODORE 64’.

, n ---

Stores n into the next available dictionary memory cell, advancing the dictionary pointer. (comma)

– n1 n2 --- diff

Leaves the difference of n1–n2.

-->

Continues interpretation with the next disc screen (pronounced next-screen).

-DUP n1 --- n1 (if zero)
n1 --- n1 n1 (if non-zero)

Reproduces 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 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.

. n ---

Prints 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 procedure to transmit the text to the selected output device. If executed outside a definition, ." will immediately print the text until the final ". See (.").

.LINE line scr ---

Prints on the screen a line of text from the RAM disc by its line and screen number. Trailing blanks are suppressed.

.R n1 n2 ---

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

/ n1 n2 --- quot

Leaves the signed quotient of n1/n2.

/MOD n1 n2 --- rem quot

Leaves 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

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

=0 n --- f

Leaves a true flag if the number is equal to zero, otherwise leaves a false flag.

0BRANCH f ---

The run-time procedure 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

Increments n1 by 1.

2+ n1 --- n2

Increments n1 by 2.

2! nlow nhigh addr ---

32 bit store. nhigh is stored at addr; nlow is stored at addr+2.

2@ addr --- nlow nhigh

32 bit fetch. nhigh is fetched from addr: nlow is fetched from addr+2.

2CONSTANT d ---

A defining word used in the form:

d 2CONSTANT cccc

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

2DROP d ---

Drops the double number from the stack.

2DUP n2 n1 --- n2 n1 n2 n1

Duplicates the top two values on the stack. Equivalent to OVER OVER

2OVER d1 d2 --- d1 d2 d1

Copies the first double value d1 to the top of the stack.

2SWAP d1 d2 --- d2 d1

Exchanges the top two double numbers on the stack.

2VARIABLE

A defining word used in the form:

d VARIABLE cccc

When VARIABLE is executed, it creates the definition cccc with its parameter field field initialised to d. When cccc is later executed, the address of its parameter field (containing d) is left on the stack, so that a double fetch or store may access this location.

:

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.

;

Terminates a colon-definition and stops further compilation. Compiles the run-time ;S.

;CODE

Used in the form:

: cccc … ;CODE

assembly mnemonics

it stops compilation and terminates a new defining word cccc by compiling (;CODE). Sets the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following mnemonics. If the ASSEMBLER is not loaded, code values may be compiled using , and C, .

When cccc later executes in the form:

cccc nnnn

the word nnnn will be created with its execution procedure 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

Stops 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 procedure.

< n1 n2 --- f

Leaves a true flag if n1 is less than n2; otherwise leaves a false flag.

<#

Set up for pictured numeric output formatting, using the words:

<# # #S SIGN #>

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

<BUILDS

Used within a colon-definition:

: cccc <BUILDS …
DOES> … ;

Each time cccc is executed, <BUILDS defines a new word with a high-level execution procedure. 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 procedures to be written in high-level rather than in assembler code (as required by ;CODE).

= n1 n2 --- f

Leaves a true flag if n1=n2; otherwise leaves a false flag.

> n1 n2 --- f

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

>R n ---

Removes a number from the computation stack and places it as the most assessible on the return stack. Its use should be balanced with R> in the same definition.

? addr ---

Prints the value contained at the address in free format according to the current base.

?COMP

Issues an error message if not compiling.

?CSP

Issues an error message if the stack position differs from value saved in CSP.

?ERROR f n ---

Issues an error message number n, if the boolean flag is true.

?EXEC

Issues an error message if not executing.

?LOADING

Issues an error message if not loading.

?PAIRS n1 n2 ---

Issues an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK

Issues an error message if the stack is out of bounds.

?TERMINAL --- f

Performs a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation.

@ addr --- n

Leaves the 16 bit contents of address.

ABORT

Clears the stacks and enters the execution state. Returns control to the operator's terminal, printing a message appropriate to the installation.

ABS n --- u

Leaves the absolute value of n as u.

AGAIN addr n --- (compiling)

Used in a colon-definition in the form:

BEGIN … AGAIN

At run-time, AGAIN forces execution to return to 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 ---

Adds 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

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

ASSEMBLER

The vocabulary that holds the FORTH assembler.

AT n1 n2 ---

Moves the cursor position to line n1, column n2.

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. However for the Commodore 64, this becomes 24 lines of 40 characters.

BACK addr ---

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

BASE --- addr

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

BEGIN --- addr n (compiling)

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 values for “blank”.

BLANKS addr count ---

Fills an area of memory beginning at addr with blanks.

BLK --- addr

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

BLOCK n --- addr

Leaves the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disc to whichever 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.

BRANCH

The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.

BORDER n ---

Sets the border to colour n.

BUFFER n --- addr

Obtains 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 block is not read from the disc. The address left is the first cell within the buffer for data storage.

C! b addr ---

Stores 8 bits at address.

C/L --- n

A constant leaving the number of characters per line; used by the editor.

C, b ---

Stores 8 bits of b into the next available dictionary byte, advancing the dictionary pointer.

C; ---

Terminates an assembler definition.

C@ addr --- b

Leaves the 8 bit contents of memory address.

CASE --- n (compiling)

Occurs in a colon definition in the form:

```
CASE
n OF … ENDOF
…
ENDCASE
```

At run-time, CASE marks the start of a sequence of OF … ENDOF statements.

At compile-time, CASE leaves n for compiler error checking.

CFA pfa --- cfa

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

CLS ---

Performs the clear screen-home cursor function.

CMOVE from to count ---

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

CODE

Creates a new word, and sets the current vocabulary to ASSEMBLER, ready for code definitions.

COLD

The cold start procedure 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

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 to the stack.

CONTEXT --- addr

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

COUNT addr1 --- addr2 n

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

CR

Transmits 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 words parameter field. The new word is created in the CURRENT vocabulary.

CSP --- addr

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

D+ d1 d2 --- dsum

Leaves the double number sum of two double numbers.

D+– d1 n --- d2

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

D. d ---

Prints a signed double number from a 32 bit two's complement value. The high-order 16 bits are most accessible on the stack. Conversion is performed according to the current BASE. A blank follows. Pronounced D-dot.

D.R d n ---

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

DABS d --- ud

Leaves the absolute value ud of a double number.

DECIMAL

Sets the numeric conversion BASE for decimal input-output.

DEFINITIONS

Used in the form:

cccc DEFINITIONS

it sets 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)

If compiling, compiles 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

Converts d1 to its double number two's complement.

DO n1 n2 --- (execute)
addr n --- (compile)

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 values 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), and 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>. It is 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

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

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.

DROP n ---

Drops the number from the stack.

DUP n --- n n

Duplicates the value on the stack.

ELSE addr1 n1 --- addr2 n2 (compiling)

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 ---

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

EMPTY-BUFFERS

Marks all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disc. This is also an initialisation procedure 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-delimiter character n1, offset to the first delimiter after the text n2, and the offset to the first character not included. This procedure will not process past an ASCII ‘null’, treating it as an unconditional delimiter.

END

This is an ‘alias’ or duplicate definition for UNTIL.

ENDCASE addr n --- (compile)

Occurs in a colon-definition in a form:

CASE n OF … ENDOF … ENDCASE

At run-time, ENDCASE marks the conclusion of a CASE statement.

At compile-time, ENDCASE computes forward branch offsets.

ENDIF addr n --- (compile)

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.

ENDOF addr n --- (compile)

Used as ENDIF but in CASE statements.

ERASE addr n ---

Clears a region of memory to zero from addr over n addresses.

ERROR line --- in blk

Executes error notification and restart of systems. 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 (RAM 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. Final action is execution of QUIT.

EXECUTE addr ---

Executes the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXPECT addr count ---

Transfers characters from the terminal to address, until a “return” or the count of characters has been received. One or more nulls are added at the end of the text.

FENCE --- addr

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 ---

Fills 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

A user variable for control of number output field width. Presently unused in fig-FORTH.

FLUSH

Writes all updated disc buffers to RAM disc.

FORGET

Executed in the form:

FORGET cccc

it deletes the definition named cccc from the dictionary with all entries physically following it. In fig-FORTH, an error message will occur if the CURRENTand CONTEXT vocabularies are not currently the same.

FORTH

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

Leaves the address of the next available dictionary location.

HEX

Sets 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

Used within a DO-LOOP to copy the loop index to the stack. See R.

I' --- n

Copies the last but one value from the return stack.

ID. addr ---

Prints a definition's name from its name field address.

IF f --- (run time)
--- addr n (compile)

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. If f is just after ELSE, it executes the false part. 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

Marks 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 an immediate definition by preceeding it with [COMPILE].

IN --- addr

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 ---

Prints 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.

INIT-DISC

Wipes all information off the RAM disc prior to first use.

INK n ---

Sets the ink colour to n (0–15)

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.

INVERSE f ---

Sets the screen to inverse if true, else sets to normal.

J --- n

Used within nested DO loops. Returns the index value of the outer loop.

KEY --- v

Leaves the ASCII value of the next terminal key struck.

LATEST --- addr

Leaves the name field address of the topmost word in the CURRENT vocabulary.

LEAVE

Forces 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

Converts 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.

LINE n --- addr

Leaves address of line n of current screen. This address will be in the disc buffer area.

LIST n ---

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

LIT --- n

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

LITERAL n --- (compiling)

If compiling, it compiles 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 ---

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

LOADR n ---

Loads the information of a RAM disc area previously saved with SAVER. n is a number from 1 to 26.

LOOP addr n --- (compiling)

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.

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.

M/MOD ud1 u2 --- u3 ud4

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 --- max

Leaves the greater of two numbers.

MESSAGE n ---

Prints 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 (RAM disc system).

MIN n1 n2 --- min

Leaves the smaller of two numbers.

MINUS n1 --- n2

Leaves the two's complement of a number.

MOD n1 n2 --- mod

Leaves the remainder of n1/n2, with the same sign as n1.

MON

Exits to Basic.

NEXT

This is the inner interpreter that uses the interpretive pointer IP to execute compiled Forth definitions. It is not directly executed but is the return point for all code procedures. It acts by fetching the address pointed by IP, 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 IP and W are computer specific. (See earlier note on FORTH conventions.)

NFA pfa --- nfa

Converts the parameter field address of a definition to its name field.

NOOP

A FORTH no-operation.

NOT f --- f

Leaves a false flag if a true flag is on the stack, else leaves a true flag. (Actually executes 0=)

NUMBER addr --- d

Converts a character string left at addr with a preceding 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

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

OR n1 n2 --- or

Leaves the bit-value logical or of two 16 bit values.

OUT --- addr

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

OVER n1 n2 --- n1 n2 n1

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

PAD --- addr

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

PFA nfa --- pfa

Converts 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

Inputs 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

Clears the return stack, stops compilation, and returns control to the operator’s terminal. No message is given.

R --- n

Copies the top of the return stack to the computation stack.

R# --- addr

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.

R> --- n

Removes the top value from the return stack and leaves it on the computation stack. See >R and R.

R0 --- addr

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

REPEAT addr n --- (compiling)

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

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

RP@ --- addr

Leaves the current value in the return stack pointer register.

RP!

A computer dependent procedure to initialise the return stack pointer from user variable R0.

S->D n --- d

Sign extends a single number to form a double number.

S0 --- addr

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

SAVER n ---

Saves a RAM disc area to tape or disc with the name @DISCn. Can be re-loaded with n is a number from 1 to 26.

SCR --- addr

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 next 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 uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SP!

A computer dependent procedure to initialise the stack pointer from S0.

SP@ --- addr

A computer dependent procedure 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

Transmits an ASCII blank to the output device.

SPACES

Transmits n ASCII blanks to the output device.

STATE --- addr

A user variable containing the compilation state. A non-zero value indicates compilation.

SWAP n1 n2 --- n2 n1

Exchanges the top two values on the stack.

TASK

A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety.

TEXT c ---

Accepts following text to PAD. c is the text delimeter.

THEN

An alias for ENDIF.

TIB --- addr

A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---

Complements the contents of addr by the bit pattern b.

TRAVERSE addr1 n --- addr2

Moves 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 addr resulting is the address of the other end of the name.

TRIAD scr ---

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

TYPE addr count ---

Transmits count characters from addr to the selected output device.

U< u1 u2 --- f

Leaves the boolean value of an unsigned less-than comparison. Leaves f=1 for u1 < u2; otherwise leaves 0. This function should be used when comparing memory addresses.

U* u1 u2 --- ud

Leaves the unsigned double number product of two unsigned numbers.

U. u ---

Prints an unsigned 16-bit number converted according to BASE. A trailing blank follows.

U.R u n ---

Prints the unsigned number u right aligned in a field whose width is n. No following blank is printed.

U/MOD ud u1 --- u2 u3

Leaves the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

UNTIL f --- (run-time)
addr n --- (compile)

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 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 its 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

The parameter field of cccc contains n as a fixed offset relative to the user pointer register UP for this user variable. When cccc is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable.

VARIABLE

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 its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VLIST

Lists the names of the definitions in the context vocabulary. “Break” will terminate the listing.

VOC-LINK --- addr

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 by FORGETting through multiple vocabularies.

VOCABULARY

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 chained so 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.

WARNING --- addr

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 user defined procedure. See MESSAGE, ERROR.

WHILE f --- (run time)
ad1 n1 --- ad1 n1 ad2 n2 (compile)

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 then 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

In fig-FORTH, it is a user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 through to 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 ---

Reads the next text characters from the input stream being interpreted, until a delimiter c is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, the characters, and ends with two or more blanks. Leading occurences 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 a pseudonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the execution procedure 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

Leaves the bitwise logical exclusive or of two values.

[

Used in a colon-definition in the form:

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

it suspends compilations. The words after [are executed, not compiled. This allows calculations or compilation exceptions before resuming compilation with]. See LITERAL,].

[COMPILE]

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 is executed, rather than at compile time.

]

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

C64

Melbourne House

Now with **C64 FORTH+** you can program your Commodore 64 with a language even more flexible than **BASIC**, yet almost as powerful as machine code.

C64 FORTH+ is a high level language that will enable you to run programs up to fifty times faster than those written in **BASIC**, without the tedious programming requirements of machine language.

You can even re-define the language to suit your own programming requirements! **C64 FORTH+** is the tool for programmers who want to squeeze as much power and speed from the Commodore 64 as possible.

As well, **C64 FORTH+** allows you to create stand-alone programs which can be run on any Commodore 64 without the need to load **C64 FORTH+** ! You can create commercial programs written in **C64 FORTH+**, without the need to pay any royalties for the use of **C64 FORTH+**.

C64 FORTH+ is a complete implementation of **fig-FORTH** with added colour, sprite, sound and graphics instructions which can create programs making full use of all the Commodore 64 features. It is also possible to define your own character set, and all of these functions can be programmed in a manner that is much easier than programming in **BASIC**.

"**FORTH** is an easy language to use, and the graphic commands in **FORTH** allow you to do anything in **FORTH** that you can do in **BASIC**. However, the extra speed of **FORTH** makes a big difference. You can get speed improvements of ten to fifty times, making it possible to produce moving graphics you would otherwise have to write in machine code."

—**PERSONAL COMPUTER NEWS**

© 1985 John Steele Jones-Steele
Published by MELBOURNE HOUSE
Castle Yard House, Castle Yard, Richmond, TW10 6TF, U.K.
70 Park Street, South Melbourne, 3205, Australia