10 What ‹BUILDS and DOES› Does

10.1 Introduction

FORTH can be considered to operate at a number of different levels. The lowest level is the execution of a word from the dictionary and this can be termed a level 0 operation. The next higher level, which we can call level 1, is the use of a defining word, e.g. VARIABLE, or <:>, to produce a dictionary entry for later (level 0) execution. All levels higher than 0 result in a new entry being made in the dictionary. This chapter is concerned with the next higher level, level 2, in which new defining words are formed. The sequence of operations which is involved is:

a) generate a new defining word (level 2),

b) use the defining word to produce a new dictionary entry (level 1),

c) execute the new entry (level 0).

One higher level is possible: to produce alternative ways of generating defining words. This level, which is often termed "meta-FORTH", enables the writing of totally new FORTH-like languages, and is beyond the scope of this manual.

Each defining word in FORTH can be considered as a mini-compiler, dedicated to compiling a particular type of structure into the dictionary. If a new structure is required, e.g. an array, a new word is required to allow its compilation. Just as the generation of a new word (level 1) extends the FORTH language, the generation of a new defining word (level 2) extends the FORTH compiler.

The two words <BUILDS and DOES> are used for this purpose. The < and > signs (which are not pronounced!!) are present to indicate that the two words should be used together and to show their order of use.

10.2 The Actions of <BUILDS and DOES>

The two words are used in a definition of the following form (level 2):

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

where an optional list of words may follow each of the two. This is, in one sense, an ordinary colon-definition and all the words are compiled in the normal way. The use of <BUILDS and DOES> , however, make it a level 2 definition.

The words following <BUILDS are concerned with building the dictionary entry for the new word defined by FAMILY . Those following DOES> determine the action of the new word, and though they are compiled into the definition of FAMILY , they are not executed until the new word is used. It is important to remember that the <BUILDS words come into effect at compilation time, and the DOES> words at execution time.

The execution of FAMILY takes the form:

```
FAMILY MEMBER
```

and may expect one or more values on the stack, depending on the

definition of FAMILY . This is a level 1 process and creates a dictionary entry for the word MEMBER .

When <BUILDS executes it generates the dictionary entry header for MEMBER , including its name, the link pointer to the previous dictionary entry and a code field pointer as normal. It also reserves an extra two-byte space immediately following the code field pointer. The words, if any, after <BUILDS then execute and will usually act to place values, or reserve space, in the parameter area of MEMBER .

The execution of DOES> takes the address of the first word after DOES> in the dictionary entry of FAMILY and places it in the two-byte space reserved by <BUILDS , creating a pointer to the list of words that will be executed when MEMBER is used. It also re-writes the contents of MEMBER's code-field address to point to a routine that will handle this execution.

When MEMBER is executed (level 0) the address of its parameter area is placed on the stack and then the words following DOES> , in FAMILY , are executed. Execution of all words defined by FAMILY begins with this code, so FAMILY produces a group of words with related actions. <BUILDS and the words following it 'customise' each new word by compiling items unique to it (e.g. values from the stack, further words) into the parameter area of its dictionary entry. When the new word is used, the words after DOES> use the address of the parameter area to gain access to these items, allowing each of the words created by the same defining word to have its own function.

10.3 The use of <BUILDS and DOES>

Some simple examples may make the use of these words more clear.

Let us first have a look at an alternative definition of VARIABLE. This word appears in the nucleus dictionary and its action has been described in Section 5.3.2. It creates a dictionary entry with space for a single variable, and initialises it. The following definition of VARIABLE is identical except that the values of the variables it creates are not initialised.

```
: VARIABLE <BUILDS 2 ALLOT DOES> ;
```

When this is executed by typing:

```
VARIABLE SIZE
```

<BUILDS creates the dictionary entry for SIZE and 2 ALLOT reserves two bytes in the parameter area. In this case there are no words after DOES> so when SIZE is executed it just leaves the address of the parameter field on the stack. This gives access to the value, which is initally indeterminate, through ! and @ as normal.

The definition of the VARIABLE in the nucleus dictionary would be:

```
: VARIABLE <BUILDS , DOES> ;
```

Instead of allotting space, the value on the top of the stack is compiled into the parameter area by <,> .

The definition of CONSTANT is:

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

The compilation stage is identical to that of VARIABLE , but when the word defined by CONSTANT is used, @ leaves the value on the stack.

We may also create single byte variables and constants by:

```
: CVARIABLE <BUILDS C, DOES> ;
```

and

```
: CCONSTANT <BUILDS C, DOES> C@ ;
```

These both expect a value in the range 0 to 255 on the stack to initialise the value of the dictionary entry they create.

10.4 Arrays and Tables

The use of <BUILDS and DOES> to create new types of data structure can be illustrated by the extension of FORTH to handle arrays.

10.4.1 One-dimensional Arrays

A simple definition for a one-dimensional array is:

```
: ARRAY
      <BUILDS   2* ALLOT
      DOES>     SWAP 2* +
;
```

A ten-element array of single precision variables is created by:

```
10 ARRAY VALUES
```

The words after <BUILDS reserve two bytes for each element. When VALUES is executed, the index is taken from the stack and converted to the address of the corresponding element. The contents of the array VALUES are not initialised but it may be filled by, for example:

```
5 0 VALUES !
10 1 VALUES !
```

which will put the numbers 5 and 10 into the first two elements of VALUES. The contents of a particular element may be placed on the stack by, for example,

```
1 VALUES @
```

or typed on the display by

```
1 VALUES ? 10 OK
```

The array index must be on top of the stack before executing VALUES. It must, for the above example, be in the range 0 to 9 inclusive. No checks are made on the range of the index so care must be taken not to over-write other dictionary entries by using an out-of-range index.

The following alternative definition of ARRAY will check the range and give an error message, if needed.

```
: ARRAY
    <BUILDS DUP 1 - ,   ( STORE MAXIMUM INDEX )
            2* ALLOT    ( RESERVE SPACE )
    DOES>   2DUP        ( DUPLICATE INDEX AND PARAMETER ADDRESS )
            @ OVER <    ( CHECK IF INDEX TOO LARGE )
```

```
            SWAP 0< OR ( OR IF NEGATIVE )
            5 ?ERROR   ( ISSUE ERROR MESSAGE IF NEEDED )
            2+         ( OTHERWISE STEP OVER MAXIMUM INDEX VALUE )
            SWAP 2* +  ( AND CONVERT INDEX TO ELEMENT ADDRESS )
;
```

If a more specific error message is required, the words 5 ?ERROR may be replaced with, for example,

```
IF    DROP CR
      ." RANGE ERROR - ARRAY INDEX = " .
      QUIT
THEN
```

The inclusion of error checks, such as that given above, has the disadvantage that it decreases the speed of execution. A solution to this problem is to develop an application using full error checks until it is working correctly. When it is certain that no errors can occur, the words containing error checks can be replaced by simpler, faster, versions. If an application is developed by use of the editing facilities described in Chapter 8, it is a simple matter to change these words as the remainder of the application is unchanged.

10.4.2 Two-Dimensional Arrays

The following definition allows the creation of two-dimensional arrays. The elements are single precision variables and the array contents are not initialised. No index checking is done but error checks could be added in a similar manner to that given for one-dimensional arrays.

```
: 2ARRAY
      <BUILDS DUP ,       ( STORE SECOND INDEX )
              * 2* ALLOT  ( RESERVE SPACE )
      DOES>   ROT         ( GET FIRST INDEX TO TOP )
              OVER @ *    ( MULTIPLY BY STORED INDEX )
              ROT +       ( ADD SECOND INDEX )
              2*          ( CALCULATE BYTE OFFSET )
              +           ( ADD TO BASE ADDRESS )
              2+          ( STEP OVER STORED INDEX )
;
```

The use is

```
10 5 2ARRAY RECTANGLE
```

to create a 10 by 5 array called RECTANGLE. In this example the array indices may range from 0,0 to 9,4 inclusive. The address of, say, the 2,3 element is left on the stack by:

```
2 3 RECTANGLE
```

10.4.3 Tables

It may be necessary to create a table of values for which only the starting address is needed. This type of structure can be implemented

very simply as follows:

```
: CTABLE
      <BUILDS ALLOT
      DOES>
;
```

This, when used in the form

```
10 CTABLE DATA
```

will create the word DATA with space for ten single-byte values. When DATA is executed it will leave on the stack the starting address of the data table.

The word LIST of Section 6.3 is an example of a table which leaves both its start address and the number of 16-bit items it contains. It may be created by use of the following definition of the word TABLE .

```
: TABLE
     <BUILDS DUP ,              ( store no. of items )
             2* ALLOT           ( reserve space )
     DOES>   DUP 2+             ( get start address )
             SWAP @             ( max number of items )
;
```

LIST is then created by

```
n TABLE LIST
```

where n is the required maximum number of items.

10.5 Strings

There are many ways of implementing string handling in FORTH. Two examples are given in 'BYTE' magazine, in the August 1980 and February 1981 issues.

The following example shows a simple alternative method to handle strings up to 255 characters in length.

```
: STRING              ( max length ... )
        <BUILDS
              DUP C,  ( KEEP MAXIMUM LENGTH )
              0  C,   ( ZERO LENGTH BYTE = EMPTY )
              ALLOT   ( RESERVE SPACE )
        DOES>
              1+      ( STEP OVER MAXIMUM LENGTH )
;
```

An empty string is then created by, for example:

```
10 STRING WORDS
```

The string variable WORDS may now hold any character string up to 10 characters in length. A few additional words are required for input and output of strings.

The definition of $IN uses the constant C/L , which gives the number of characters per line in the display (i.e. 64). Remember also that PAD returns the start address of the scratchpad area used for text (Section 8.2.3) and for numeric conversion (Section 7.2.4).

```
: $IN                              ( ... addr\length )
     HERE C/L 1+ BLANKS            ( CLEAR MEMORY AT HERE )
     1 WORD                        ( INPUT STRING TO HERE )
                                   ( TERMINATED BY CARRIAGE RETURN )
     HERE PAD C/L 1+ CMOVE         ( MOVE STRING TO PAD )
     PAD DUP C@ 1+                 ( PREPARE TO MOVE STRING ... )
;                                  ( INCLUDING COUNT BYTE )

: $!                               ( from addr\length\to addr ... )
     2DUP 1 - C@ 1+ >              ( CHECK IF SPACE FOR STRING )
     IF CR ." STRING OVERFLOW "    ( IF NOT GIVE ERROR )
        2DROP DROP QUIT            ( CLEAR STACK & QUIT )
     THEN
     SWAP CMOVE                    ( OTHERWISE STORE STRING )
;

: $@                               ( addr1 ... addr2\length )
     COUNT                         ( PREPARE TO TYPE STRING )
;
```

The following shows how these words are used, assuming that the string variable WORDS has been created as in the earlier example.

```
$IN HELLO OK
WORDS $! OK
WORDS $@ TYPE SPACE HELLO OK
```

If the words LEFT$ and RIGHT$ of Section 7.3.2 are also defined, the following examples can be tried.

```
WORDS 2 LEFT$ TYPE SPACE HE OK
WORDS 3 RIGHT$ TYPE SPACE LLO OK
```

10.6 A CASE Statement

10.6.1 Introduction

The conditional structure of Section 6.2 allows a two-way branch using

```
IF ... ELSE ... THEN
```

A CASE statement allows a branch to one of many possible word sequences with a return to a common point. There are two basic methods for the selection of the case to be executed. The first is a 'positional' case where the values to be tested are restricted to the first n integers. The second method is a 'keyed' case where a value is tested against a sequence of explicit values which need not be in numerical order.

10.6.2 A Positional CASE

The following simple example of a positional CASE will select the words to be executed by means of an integer value on the stack. The value must be in the range from zero to one less than the number of cases available in the particular example. No error checks are made for a number outside the permitted range. This CASE structure is used in the graphics package to select the resolution mode, and has an added check to restrict the choice to the integers from 0 to 3 inclusive.

Here is the definition of the defining word CASE: :

```
: CASE:
       <BUILDS SMUDGE ]
        DOES>  SWAP 2*
               + @
               EXECUTE
;
```

The word EXECUTE takes the execution (code field) address of a word from the stack and executes the word's definition. Thus

```
' WARM    ( get parameter field address of WARM )
CFA       ( convert to code field address )
EXECUTE
```

has the same effect as executing WARM directly from the keyboard.

To use the case structure it is first necessary to define each of the possible actions, for example:

```
: NOTHING
   ." CASE 0 DOESN'T DO MUCH " ;

: BELL
   ." CASE 1 RINGS THE BELL "
   7 EMIT ;

: HOME
   ." CASE 2 HOMES THE CURSOR "
   30 EMIT ;
```

These actions are then included in a case structure for, say, the word TEST .

```
CASE: TEST
       NOTHING  BELL  HOME  ;
```

When TEST is being created, SMUDGE ensures that the new entry will be found in a dictionary search and] then sets compilation mode, so that the words following TEST will have their addresses compiled into the dictionary entry.

When TEST is executed by:

```
0 TEST
1 TEST
or
2 TEST
```

the words following DOES> convert the case number to a pointer to the address of the correct word in the list, and execute it.

Note that the CASE statements of many high-level languages are based on GOTO-type control transfers, whereas this FORTH CASE has the options compiled into the definition of the case word so that the choice is fixed before execution. Basically, this is because it is not easy to handle forward references, i.e. words that have not yet been defined, in FORTH.

For a further discussion of a variety of possible forms for case statements in FORTH see 'FORTH Dimensions' Vol.2 No.3 (1980) (see Appendix E).

11 Further Examples

11.1 Fast Divide-by-Two

Division reqires a large number of operations and is usually very slow in a microcomputer. The following routine is a machine code primitive which will divide by two. Its action is identical with

```
2 /
```

but is about sixty times faster.

```
HEX

CREATE 2/
    18  C,  1B5  ,  910  ,  F6   ,
    2D0  ,  1F6  ,  1F0  ,  38  C,
    176  ,   76  ,  4C  C,  2842 ,
SMUDGE
```

Assembly Listing

```
18                      CLC
B5 01                   LDA 1,X
10 09                   BPL 2D1V
F6 00                   INC 0,X
D0 02                   BNE NOINC
F6 01                   INC 1,X
F0 01          NOINC    BEQ 2DIV
38                      SEC
76 01          2DIV     ROR 1,X
76 00                   ROR 0,X
4C 42 28                JMP NEXT
```

11.2 Recursion

A recursive routine is one which uses itself. In FORTH a dictionary entry cannot find a reference to itself while it is being defined. This problem is solved by defining the IMMEDIATE word MYSELF.

```
: MYSELF
     LATEST          ( get name field address of latest word )
     PFA             ( convert to parameter field address )
     CFA             ( convert this to code field address )
     ,               ( compile the address )
;    IMMEDIATE
```

11.2.1 Factorials

The following example shows its use to form a recursive definition to calculate factorials.

```
: (FACT) ( nl\n2 ... n3 )
      -DUP IF DUP ROT * SWAP
              1 - MYSELF
           THEN
;

: FACT ( n ... )
      DUP 0< OVER 7 > OR 5 ?ERROR
      1 SWAP (FACT) .
;
```

The calculation of the factorial is performed by (FACT), which leaves the result on the stack. Attempting to calculate factorials of numbers greater than 7 will cause an arithmetic overflow. Factorials of negative numbers are not defined. Error checking is confined to FACT. This then uses (FACT) to calculate the result, which is displayed.

The following, alternative, definition of FACT makes use of MD*, defined in Section 4.4. The result is left as a double precision number, allowing the calculation of factorials of numbers up to and including 12.

```
: (FACT)
      -DUP IF DUP 1 -
              >R MD* R>
              MYSELF
           THEN
;

: FACT ( n ... )
       DUP 0< OVER 12 > OR 5 ?ERROR
       1 0 ROT (FACT) D.
;
```

11.2.2 Sorting an Array

The word QUICKSORT (nl\n2 ...) uses the quicksort algorithm to sort elements nl to n2 inclusive of the array NUMBERS into increasing numerical order.

Before typing in the following definitions you must first define

```
2/        ( Section 11.1 )
ARRAY     ( Section 10.4.1 )
MYSELF    ( Section 11.2 )

0 VARIABLE TEMP                    ( TEMPORARY STORAGE )
256 ARRAY NUMBERS                  ( OR WHATEVER SIZE YOU WISH )

: EXCHANGE ( nl\n2...)             ( EXCHANGE ELEMENTS nl & n2)
                                   ( OF NUMBERS )
  NUMBERS DUP >R @ SWAP
  NUMBERS DUP @ >R
  ! R> R> !
;

: PARTITION ( nl\n2 ... n3\n4 )    ( SPLIT ARRAY INTO SMALLER SECTIONS,
                                   EXCHANGING ELEMENTS WHERE NECESSARY )
  BEGIN 2DUP > 0= WHILE
  SWAP BEGIN DUP NUMBERS @ TEMP @ < WHILE 1+ REPEAT
  2DUP EXCHANGE SWAP 1+ SWAP 1 -
  REPEAT
```

```
;

: QUICKSORT ( nl\n2 ... )
    2DUP 2DUP + 2/ NUMBERS @ TEMP ! PARTITION
    >R ROT DUP R < R> SWAP
    IF      MYSELF ELSE 2DROP THEN 2DUP >
    IF SWAP MYSELF ELSE 2DROP THEN
;
```

The routine can be checked by filling NUMBERS with integers in reverse order by, for example

```
: NFILL ( n ... )
   DUP 0 DO
          DUP I NUMBERS ! 1 -
       LOOP DROP
;
```

Then execute, for example,

```
100 NFILL
```

which should fill the first 100 elements of NUMBERS with reverse-order integers. Executing

```
0 99 QUICKSORT
```

should then leave the elements of NUMBERS in ascending order. The contents can be checked by, for example:

```
: NCHECK ( n ... ) 0 DO I NUMBERS ? LOOP ;
```

Then execute, for example:

```
100 NCHECK
```

11.3 A Screen Copying Utility

Copying screens, particularly with a tape-based version, can be very tedious. The following routines will allow the copying of up to five screens at a time from one tape to another. The screen contents are temporarily stored in the graphics memory from #8200 to #8BFF. The screens may be renumbered during the copying process but the input and output screens must be numbered consecutively. They require the prior loading of the tape interface.

```
BASE @                  ( SAVE CURRENT BASE )
HEX

: OFFKEY                ( DISABLE KEYBOARD )
    -1 20A +! ;

: ONKEY                 ( RE-ENABLE KEYBOARD )
     1 20A +! ;

: PAUSE                 ( WAIT FOR A KEYPRESS )
     KEY DROP CR ;

: INSCR ( first screen no.\no. of screens to input ... )
     CR ." PLAY " PAUSE
```

```
      OFFKEY
      0 DO
         DUP I + LIST      ( LOAD AND LIST SCREEN )
         FIRST 200 DUP
         I * 8200 +
         SWAP CMOVE        ( MOVE TO GRAPHICS MEMORY )
      LOOP DROP
      ONKEY
;

: OUTSCR ( first screen no.\no. of screens to output ... )
      OVER SCR !           ( SET NEW SCREEN NO. )
      ." RECORD" PAUSE
      OFFKEY
      0 DO
         I 200 * 8200 +    ( TRANSFER FROM    )
         FIRST 200 CMOVE   ( GRAPHICS MEMORY )
         SAVE
         1 SCR +!          ( INCREMENT SCREEN NO. )
         LOOP DROP
      ONKEY
      -1 SCR +!            ( RESET FOR NEXT BATCH )
;

: COPY ( first input screen no.\first output screen no.\
                           no. of screens ... )
      5 MIN                ( ENSURE NOT MORE THAN 5 SCREENS )
      >R SWAP R            ( KEEP NO. OF SCREENS ON RETURN STACK )
      INSCR
      R>                   ( RECOVER NO. OF SCREENS )
      ." NEW TAPE " PAUSE
      OUTSCR
;

BASE !                     ( RESTORE ORIGINAL BASE )
```

To copy screens 6, 7 and 8 to a new tape, numbered as 15, 16 and 17, execute:

```
6  15  3  COPY
```

Separate use of INSCR and OUTSCR will allow intermediate editing of the screens provided each screen is moved from the graphics memory to the tape buffer, edited and then returned to the same area of the graphics memory. In HEX,

```
        n FIRST 200 CMOVE ( graphics -> tape buffer )
      FIRST  n  200 CMOVE ( tape buffer -> graphics )
```

where n may be

8200, 8400, 8600, 8800 or 8A00,

depending on which of the 5 screens is to be edited.

11.4 Use of the Operating System Monitor Routines

It may be necessary, from within FORTH, to use the cassette operating system commands. The following definitions allow this to be done either by direct execution from the keyboard or from within a definition.

```
BASE @ HEX

CREATE OSCLI ( call the OSCLI routine of the monitor )
      8E86 , 20 C, FFF7 ,
      8EA6 , 4C C, 2842 ,
SMUDGE

: (MONITOR)
    1 WORD HERE PAD C/L 1+ CMOVE  ( TRANSFER KEYED INPUT TO PAD )
    PAD COUNT                     ( GET ADDRESS & LENGTH OF STRING )
    2DUP + 0D SWAP C!             ( ADD 'RETURN' TO END )
    1+                            ( INCREASE STRING COUNT )
    100 SWAP CMOVE CR             ( TRANSFER TO COS BUFFER )
    IN @ 60 IN !                  ( PUT RTS HERE FOR MONITOR )
    OSCLI                         ( INTERPRET & EXECUTE )
    IN !                          ( RESTORE VALUE OF IN )
;

: MONITOR
     STATE @ IF                   ( COMPILING A DEFINITION )
               COMPILE QUERY
               COMPILE
             THEN
  (MONITOR)
; IMMEDIATE

BASE !
```

Note that THEN does not create a compiled address but only marks the end of a conditional, for calculation of an offset. The second COMPILE will, therefore, compile (MONITOR) and not THEN.

Any of the Cassette Operating System commands of Chapter 19 of 'Atomic Theory and Practice' pages 139 to 142 may then be used. Note that the initial * of these commands should not be used. For example, to use the COS command *CAT to obtain a catalogue of a tape, the sequence:

```
MONITOR CAT
```

should be used (and not MONITOR *CAT).

If MONITOR is used in a definition, it will, on execution of the definition, wait for the command to be entered from the keyboard. This allows the command to be selected at execution time rather than being fixed at the time of definition.

11.5 WAIT

The following example is an implementation of the WAIT instruction of ATOM BASIC, which waits until the next 'tick' of the 60Hz sync signal. Each tick is signalled by a zero in the most significant bit of Port C of the 8255 PIA, at address #B002 in the ATOM.

A simple implementation is as follows:

```
HEX

: WAIT
     BEGIN B002 C@ 80 < UNTIL
     BEGIN B002 C@ 7F > UNTIL ;

DECIMAL
```

This is equivalent to the machine code subroutine at #FE66 in ATOM BASIC.

11.6 Using the Internal Loudspeaker

11.6.1 Generating Tones

The simplest way of generating a tone from the internal loudspeaker is to TOGGLE bit 2 of the output port at address #B002. This is done by the word BLIP1 which is used by TONE1 to generate a short note.

```
HEX

: BLIP1
    B002 4 TOGGLE ;

: TONE1
    100 0 DO BLIP1 LOOP ;
```

A second tone may be generated by toggling the speaker twice within a definition.

```
: BLIP2
   B002 4 2DUP TOGGLE TOGGLE ;

: TONE2
   100 0 DO BLIP2 LOOP ;
```

The words BOP and BIP are useful for sound effects for paddle-type games:

```
: BOP
   30 0 DO BLIP1 LOOP ;

: BIP
   30 0 DO BLIP2 LOOP ;
```

Finally, we can produce a warble tone by:

```
: WARBLES
   0 DO BIP BOP LOOP;
```

This is used by typing, for example:

```
10  WARBLES
```

11.6.2 Music

The following definitions will allow the keyboard to be used to play music via the internal speaker.

```
CREATE NOTE ( 0\FRE\LEN ... )
   EAEA ,   EAEA ,   6A0 ,    88 C,
   FDD0 ,    18  C,  2B5 ,   475 ,
    495 ,    3B5 ,   575 ,   595 ,
    890 ,    4A9 ,   4D  C, B002 ,
    8D  C,  B002 ,   5B0 ,  EAEA ,
   EAEA ,    EA  C,   B5 ,   5D0 ,
    1D6 ,    18  C,  590 ,  EAEA ,
   EAEA ,    EA  C,   D6 ,  C7D0 ,
    1B5 ,   C6D0 ,  E8E8 ,    4C C,
   291E ,
SMUDGE

DECIMAL

: PITCH  ( FREQUENCY ... FRE )
     256 5 */ ;

: TONE   ( FREQUENCY ... )
     PITCH 0 SWAP 2500 NOTE ;

: TABLE  ( BYTESIZE ... )
    <BUILDS ALLOT DOES> ;

20 TABLE KEYS 40 TABLE FREQUENCIES

: CFILL ( BYTEVALS\TABLENAME\BYTELENGTH ... )
   0 DO
      DUP I +
      >R SWAP R> C!
   LOOP DROP
;

   32 93 91 64 59 80 76 75
73 74 85 72 89 71 70 82 68
69 83 65   KEYS 20 CFILL

: TFILL  ( VALS\TABLENAME\LENGTH ...)
   0 DO
     DUP I 2* +
     >R SWAP R> !
   LOOP DROP
;


  0  683  640  608  576  542  512  483
456  457  406  387  352  341  320  304
288  271  256  243  FREQUENCIES  20  TFILL

: KBD ( C\ADDR ... OFFSET )
   0 BEGIN
      >R 2DUP
      R 19 > IF CR ." ?" 2DROP 1
             ELSE R + C@ =
             THEN R> 1+ SWAP
```

```
      UNTIL
   1 -
;

:  KEYBOARD
     BEGIN
       KEY KEYS KBD
       >R 2DROP R>
       2* FREQUENCIES + @ TONE
     ?ESC UNTIL
;
```

The tones produced by NOTE vary in timbre as well as pitch and are not 'pure' unless the value of FRE is an integral power of two. This is because the speaker is only toggled when the value third on the stack overflows. This occurs at slightly irregular intervals unless the above condition is met. One effect is an apparent shift in the pitch of some notes. The values in FREQUENCIES are therefore not exactly those expected for a true scale, but are chosen for the best perceived scale.

Executing KEYBOARD allows the notes to be played. Pressing the ESC key will terminate execution. The arrangement of the keys used is as follows:

11.7 Using the VIA Timer

Execution times of FORTH words can be found by use of the VIA timer as shown in the following examples. To avoid timing the compilation of the words as well as their execution the timing should be carried out within a colon-definition.

```
HEX

: TIMER-ON      ( START TIMER )
     0 B80A ! FFFF  B808 ! ;

: .TIME ( adjustment ... )  ( DISPLAY ELAPSED TIME )
      20  B80A !
     FFFF B808 @ -
     SWAP -
     . ." MICROSECONDS "
;

DECIMAL
```

The word .TIME, which displays the time interval in microseconds from the instant the timer was started, requires a time adjustment on the stack to correct for the time taken to read the counter and for any other words whose timing is not required. The time to read the counter is 620 microseconds.

Examples:

```
: DROPTEST
     620 0 TIMER-ON DROP .TIME ;

     DROPTEST  48 MICROSECONDS  OK
```

(i.e. the word DROP executes in 48 microseconds)

```
: DUPTEST
     620 TIMER-ON DUP .TIME DROP ;

     DUPTEST  71 MICROSECONDS  OK


:  CONTEST
     668 TIMER-ON 0 DROP .TIME ;

   CONTEST  77 MICROSECONDS  OK
```

This last example gives the execution time of a constant, in this case the constant 0 . Its value must be dropped from the stack before executing .TIME so the execution time of DROP is added to the time adjustment, giving a total adjustment of 668 microseconds.

The following table gives execution times for some of the more common words.

WORD:	TYPE:	EXECUTION TIME (MICROSECONDS) :
DROP	CODE	48
DUP	CODE	71
OVER	CODE	71
ROT	FORTH	424
PICK	FORTH	1168
ROLL	FORTH	9481 + 1636(n-4) (n = 4, 5, 6 ...) (NOTE: 1 ROLL = NOOP, 2 ROLL = SWAP, 3 ROLL = ROT)
@	CODE	80
!	CODE	84
DO ... LOOP	CODE	80 + 116n (n = no. of loops)
I	CODE	79
*	CODE	759 to 1379 depending on the number of non-zero bits in the multiplier
/	CODE	4230 to 4394 largely depending on the number of digits in the result.

The words * and / are colon definitions but most of the execution time is spent in the machine-code primitives U* and U/ .The relatively long execution time for ROLL is due to it being (like PICK) a FORTH definition, rather than a machine code definition, and including an error check. It is particularly slow since it involves a fairly high degree of movement of the stack contents.

11.8 Further Graphics

The following application will plot a figure similar to that shown on the front cover of this manual. It uses recursive calls to 3SIDES to plot three sides of a rectangle, their orientation being controlled by a case statement (see Section 10.6.2).

The application requires the GRAPHICS package to have been loaded previously. This contains the definition of CASE: . The variables X and Y are defined in the FORTH vocabulary and are therefore distinct from the X and Y in the GRAPHICS vocabulary. Note that this duplication is not detected by the system since, when FORTH is the CONTEXT vocabulary, no other vocabularies are searched.

```
FORTH DEFINITIONS DECIMAL

4 CONSTANT N    128 CONSTANT H0
0 VARIABLE X      0 VARIABLE Y
0 VARIABLE X0     0 VARIABLE Y0
0 VARIABLE H

: YSCALE  ( SCALE Y VALUE TO FILL SCREEN )
    3 2 */  ;

: XYLINE  ( DRAW LINE TO X,Y*3/2 )
    X @ Y @ YSCALE GRAPHICS LINE  ;

: X+      ( LINE IN +VE X-DIRECTION )
    H @ X +! XYLINE  ;

: Y+      ( LINE IN +VE Y-DIRECTION )
    H @ Y +! XYLINE  ;

: X-      ( LINE IN -VE X-DIRECTION )
    H @ MINUS X +! XYLINE  ;

: Y-      ( LINE IN -VE Y-DIRECTION )
    H @ MINUS Y +! XYLINE  ;

: 3SIDES  ( INDEX\CASE NO ... INDEX )
    OVER DUP          ( 2 COPIES OF INDEX )
    IF                ( NON-ZERO INDEX )
       1 - SWAP       ( DECREMENT INDEX AND )
                      ( BRING CASE NO. TO TOP )

       [ HERE H ! ] NOOP
                      ( RESERVE SPACE FOR ORIENTATION )
                      ( - DEFINED LATER - AND SAVE )
                      ( ADDRESS IN H )
    ELSE 2DROP        ( CASE NO. AND INDEX )
    THEN
;

: ORA ( THESE DETERMINE THE 4 ORIENTATIONS )
    3 3SIDES X-
    0 3SIDES Y-
    0 3SIDES X+
    1 3SIDES DROP
;
```

```
: ORB
    2 3SIDES Y+
    1 3SIDES X+
    1 3SIDES Y-
    0 3SIDES DROP
;

: ORC
    1 3SIDES X+
    2 3SIDES Y+
    2 3SIDES X-
    3 3SIDES DROP
;

: ORD
    0 3SIDES Y-
    3 3SIDES X-
    3 3SIDES Y+
    2 3SIDES DROP
;

CASE: ORIENTATION
    ORA ORB ORC ORD
;

' ORIENTATION CFA H @ !
    ( PLACE ADDRESS OF ORIENTATION IN 3SIDES )
    ( TO COMPLETE RECURSION )

: INITIALISE
    H0 DUP H !
    2 / DUP X0 ! Y0 !
    GRAPHICS 3 CLEAR WHITE
;

: XYSET ( START POSITION AND SIZE FOR EACH PLOT )
    H @ 2 / DUP H !
    2 / DUP X0 +! Y0 +!
    X0 @ Y0 @
    2DUP Y ! X !
    YSCALE GRAPHICS MOVE
;

: PLOT-IT
    INITIALISE
    0 BEGIN 1+ ( INCREMENT INDEX )
         XYSET
         0 3SIDES
         DUP N =
      UNTIL
    DROP
    KEY DROP 12 EMIT
;
```

When the application has been entered the figure is displayed by executing PLOT-IT. The number of iterations is governed by the constant N . Its value may be changed by, for example:

```
3 ' N !
```

In graphics mode 3, the largest value of N for a clear display is 5. However an interesting textured effect can be produced by changing the WHITE in INITIALISE by, for example:

```
GRAPHICS ' INVERT CFA ( EXECUTION ADDRESS OF INVERT )
' XYSET NFA 4 -       ( LOCATION OF WHITE IN INITIALISE )
!
```

and executing PLOT-IT with N = 6.

11.9 Non-destructive Stack Print

The following short application will allow the stack contents to be displayed without destroying them. It is useful, for example, in the development and testing of an application.

It requires the previous loading of 2/ (Section 11.1). Alternatively the somewhat slower <2 /> may be used. In addition the silent user variable S0 must be given a dictionary header by

```
6 USER S0
```

The definitions are as follows:

```
: DEPTH ( ... n ) ( returns the number of stack items )
   SP@ S0 @ SWAP - 2/ ;

: .S ( ... )      ( non-destructive stack display )
   CR DEPTH
   IF S0 @ 2 - SP@ SWAP
      DO I ? -2 +LOOP
   ELSE ." EMPTY "
   THEN
;
```

The stack items are printed with the top stack item on the right.

12 Error Messages

Most detected errors in ATOM FORTH result in an error message of the form:

```
? cccc MSG # n
```

where cccc is the word where FORTH thinks the error has occurred. The general rule in error handling is that both the return and computation stacks are cleared. The one major exception is error message 4, indicating the redefinition of an existing word, when the message is simply a warning. The message number is printed in the current base so may not be immediately recognisable. In the following error message list the message number is given in decimal and hex.

DECIMAL	HEX	Message
0	0	Unrecognised word or invalid character
1	1	Empty stack
2	2	Dictionary full
3	3	Has incorrect address mode (Assembler)
4	4	Not unique (warning only)
5	5	Index or parameter outside valid range
6	6	Tape/disc screen number out of range
7	7	Full stack
8	8	(Reserved for disc use)
9	9	)
10	A	)
11	B	) User definable
12	C	)
13	D	)
14	E	)
15	F	(Reserved for disc use)
16	10	(Reserved for disc use)
17	11	Compilation only
18	12	Execution only
19	13	Conditionals not paired
20	14	Definition not finished
21	15	In protected dictionary
22	16	Use only when loading
23	17	Off current editing screen
24	18	Declare vocabulary

Glossary of FORTH Words

This glossary contains all words present in ATOM FORTH. Each entry is of the following form:

Word Stack Action Uses Leaves Status Pronunciation

followed by a description and in many cases a numerical example. The computation (parameter) stack action is shown, where appropriate, as a list of the values and their types before and after the execution of the word, in the form:

(stack contents before ... stack contents after)

In all references to the stack, numbers to the right are at the top of the stack. The notation n1\n2 is read as "n1 is beneath n2". The symbols used to represent the different stack value types include:

n	16-bit (single precision) signed number
u	16-bit (single precision) unsigned number
addr	16-bit address (unsigned)
nd	32-bit (double precision) signed number
ud	32-bit (double precision) unsigned number
b	8-bit one-byte number (unsigned)
c	7-bit ASCII character
count	6-bit string length count
f	boolean flag: 0 = false, non-zero = true
ff	boolean false flag = 0
tf	boolean true flag = non-zero

The number of stack values that the word uses and leaves are also shown. Some words have an additional letter indicating their status.

A	only for the Acorn ATOM - not a standard FORTH word
C	may only be used in a colon definition
E	intended for execution only
P	has precedence bit set; will execute even when in compile mode

Where not obvious, the standard pronunciation is given in square brackets after the status.

The glossary contains all words that are immediately available to the user when the basic system is loaded. This includes headerless entries (see Appendix C), which are listed with their code field (execution) addresses. Since they have no names in the dictionary their names are completely arbitrary, but the names given are those preferred in a standard FORTH system.

! **(n\addr ...)** **2 0** **[store]**

Stores the value n at the address addr.

```
before: 7 35 -1234  4128
after:  7 35
```

(-1234 is stored in the two bytes from address 4128)

!CSP **0 0** **[store C-S-P]**

Stores the stack pointer value in user variable CSP. Used as part of the compiler security.

**(ndl ... nd2)** **2 2** **[sharp]**

Converts the least-significant digit (in the current base) of the double-precision number ndl to the corresponding ASCII character, which it then stores at PAD . The remaining part of the number is left as nd2 for further conversions. # is used between <# and #> .

If BASE is DECIMAL

```
before:  9 32 1234567
after:   9 32 123456
```

(ASCII code #37 is stored in PAD)

#> **(nd ... addr\count)** **2 2** **[sharp-greater]**

Terminates numeric output conversion by dropping the double number nd and leaving the address and character count of the converted string in a form suitable for TYPE .

```
.Hbefore: 23 19 0    0
after:  23 19 4128 3
```

(TYPE would display the 3-character string starting at address 4128)

#S **(ndl ... nd2)** **2 2** **[sharp-S]**

Converts the double-precision number ndl into ASCII text by repeated use of # , and stores the text at PAD . The double-precision number nd2 is left on the stack, and has a value of zero. #S is used beween <# and #> .

```
before: 27 5 1234567
after : 27 5 0000000
```

(ASCII codes #37, #36, #35, ... #31 are in consecutive memory locations at PAD)

' **(... addr)** **0 1 P** **[tick]**
(during execution)
0 0
(during compilation)

Used in the form ' nnnn and leaves the parameter field address of dictionary word nnnn if in execution mode.

If used within a colon definition it will execute to compile the address as a literal numerical value in the definition.

(**P** **[paren]**

Used in the form (nnnn) to insert a comment. All text nnnn up to a right parenthesis on the same line is ignored. Since (is a FORTH word it must be followed by a space. A space is not necessary before) since it is only used as a delimiter for the text.

) is pronounced "close-paren".

(+LOOP) **(n ...)** **1 0** **[bracket-plus-loop]**

Headerless code; execution address #28E0. The run-time procedure compiled by +LOOP that increments the loop index by the signed quantity n and tests for loop completion. See +LOOP .

(.") **[bracket-dot-quote]**

Headerless code; execution address #31E4. The run-time procedure compiled by ." that transmits the following in-line text to the output device. See ." .

(;CODE) **C**

Headerless code; execution address #3146. The run-time procedure that rewrites the code field address of the most-recently defined word to point to the machine-code following (;CODE) . It is used by the system defining words (<:>, CONSTANT etc.) to define the machine-code actions of dictionary entries using them.
This is, in a sense, a machine-code version of DOES> .

(ABORT) **[bracket-abort]**

Headerless code; execution address #347A. Executes after an error when WARNING is -1. It normally causes the execution of ABORT but the contents of the parameter area can be changed (with care) to point to to a user-defined error-handling procedure.

(DO) **C**

Headerless code; execution address #2910. The run-time procedure compiled by DO that moves the loop control parameters to the return stack. See DO .

(ENTER) **0 0** **[bracket-enter]**

Headerless code; execution address #3AF5. Interprets the current contents of the tape input buffer.

(FIND) **(addr1\addr2 ... pfa\b\tf)** **2 3** **[bracket-find]**
(found)
(addr1\addr2 ... ff) **2 1**
(not found)

Headerless code; execution address #2955. Searches the dictionary starting at the name field address addr2 for a match with the text starting at addr1. For a successful match the parameter field address and length byte of the name field plus a true flag are left. If no match is found only a false flag is left.

(LOAD) **(addr\f ...)** **2 0 A** **[bracket-load]**

Headerless code; execution address #3B69. The implementation-dependent routine used by LOAD and --> to load screens from tape or disc. The addr is that of the zero-page data required by the operating system. This data is completed by the creation of a 3-digit file name from the screen number and the insertion of its address as the first item. If the rest of the data is created by OSDATA then addr must be #62. The flag determines the appearance of the prompt on the display. In all cases an indication is given of the screen number for which a search is being made. If the flag is false a further prompt PLAY TAPE with a wait for a keypress is given. If the flag is true (non-zero) these further actions do not occur.

(LOOP) **[bracket-loop]**

Headerless code; execution address #28BA. The run-time procedure compiled by LOOP that increments the loop index by one and tests for loop completion. See LOOP .

(NUMBER) **(nd1\addr1 ... nd2\addr2)** **3 3** **[bracket-number]**

Headerless code; execution address #33B7. Converts the ASCII text beginning at addr1 + 1 according to the current numeric conversion base. The new number is accumulated into nd1, being left as nd2. Addr2 is the address of the first non-convertable digit. (NUMBER) is used by NUMBER .

***** **(n1\n2 ... n3)** **2 1** **[times]**

Leaves as n3 the product of the two signed numbers n1 and n2.

```
before: 7 -3  9
after:  7 -27
```

***/** **(n1\n2\n3 ... n4)** **3 1** **[times-divide]**

Leaves as n4 the value n1*n2/n3. The product n1*n2 is kept as a double precision intermediate value, resulting in a more accurate result than can be obtained by the sequence n1 n2 * n3 / .

```
before: 7  3  17  5
after:  7 10
```

***/MOD** **(n1\n2\n3 ... n4\n5)** **3 2** **[times-divide-mod]**

Leaves, as n4 and n5 respectively, the remainder and the integer value of the result of n1*n2/n3. The product n1*n2 is kept as a double precision intermediate value, resulting in a more accurate result than can be obtained by the sequence n1 n2 * n3 /MOD .

```
before: 7  3  17  5
after:  7  1  10
```

+ **(n1\n2 ... n3)** **2 1** **[plus]**

Leaves as n3 the sum of n1 and n2.

```
before:  19  7  24
after:   19 31
```

+! **(n\addr ...)** **2 0** **[plus-store]**

Adds n to the value at addr.

```
before: 25  -2  8427 (addr 8427 contains 29, for example)
after:  25           (addr 8427 now contains 27)
```

+- **(nl\n2 ... n3)** **2 1** **[plus-minus]**

Leaves as n3 the result of applying the sign of n2 to nl.

```
before: 17  4  -7
after:  17 -4
```

+LOOP **(n ...)** **1 0 P,C** **[plus-loop]**

Used in colon definition in the form:

DO ... +LOOP

During execution +LOOP controls branching back to the corresponding DO , dependent on the loop index and loop limit. The loop index is incremented by n, which may be positive or negative. Branching to DO will occur until

a) for positive n, the loop index is greater than or equal to the loop limit, or

b) for negative n, the loop index is less than or equal to the loop limit.

Execution then continues with the word following +LOOP .

+ORIGIN **(n ... addr)** **1 1** **[plus-origin]**

Leaves the address of the nth byte after the start of the boot-up parameter area. Used to access or modify the boot-up parameters.

, **(n ...)** **1 0** **[comma]**

Stores (compiles) n in the first two available bytes at the top of the dictionary and increments the dictionary pointer by two.

- **(nl\n2 ... n3)** **2 1** **[subtract]**

Leaves as n3 the difference nl - n2.

--> **P** **[next screen]**

Continues interpretation with the next screen of source code from tape.

-DUP **(ff ... ff)** **1 1** **[dash-dup]**
or **(tf ... tf\tf)** **1 2**

Duplicates the top stack value if it is true (non-zero).

-FIND **(... pfa\b\tf)** **0 3** **[dash-find]**
(if found)
(... ff) **0 1**
(if not found)

Used as -FIND nnnn . The CONTEXT and then the CURRENT vocabularies are searched for the word nnnn . If found, the entry's parameter field address, name length byte, and a true flag are left; otherwise just a false flag is left.

-TRAILING (addr\nl ... addr\n2) 2 2 [dash-trailing]

Changes the character count nl of the text string at addr so as not to include any trailing blanks, and leaves the result as n2.

. (n ...) 1 0 [dot]

Prints the number n on the terminal device in the current numeric base . The number is followed by one blank space.

." P [dot-quote]

Used as ." cccc"

In a colon definition the literal string cccc is compiled together with the execution address of a routine to transmit the text to the terminal device.

In the execution mode the text up to the second " will be printed immediately.

.R (nl\n2 ...) 2 0 [dot-R]

Print the number nl at the right-hand end of a field of n2 spaces. Unlike <.> no following space is printed.

/ (nl\n2 ... n3) 2 0 [divide]

Leaves the value n3 = nl / n2.

```
before: 13  27  6
after:  13  4
```

/MOD (nl\n2 ... n3\n4) 2 2 [divide-mod]

Leaves the remainder n3 and quotient n4 of nl/n2. The remainder has the sign of the dividend.

```
before:  13  27  6
after:   13   3  4
```

0,1,2 (... n) 0 1

These often-used numerical values are defined as constants in the dictionary to save both time and dictionary space.

0< (n ... f) 1 1 [zero-less]

Leaves a true flag if n is less than zero, otherwise leaves a false flag.

0= (n ... f) 1 1 [zero-equals]

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

0BRANCH (f ...) 1 0 [zero-branch]

Headerless code; execution address #28A2. The run-time procedure to cause a conditional branch. If f is false the following in-line number is added to the interpretive pointer to cause a forward or backward branch. It is compiled by IF , UNTIL and WHILE .

1+ (nl ... n2) 1 1 [one-plus]

Increments nl by one to give n2.

2* **(nl ... n2)** **1 1** **[two-times]**

Multiplies nl by two to give n2. Faster in execution than 2 * .

2+ **(nl ... n2)** **1 1** **[two-plus]**

Increments nl by two to give n2.

2DROP **(nd ...)** **2 0** **[two-drop]**

Drops the double-precision number nd (or two single precision numbers) from the stack.

2DUP **(nd ... nd\nd)** **2 4** **[two-dup]**

Duplicates the top double-precision number (or the top two single-precision numbers) on the stack.

: **P,E** **[colon]**

Used to create a colon definition in the form

: cccc .... ;

Creates a dictionary entry for the word cccc as being equivalent to the sequence of FORTH words until the next <;>. Each word in the sequence is compiled into the dictionary entry, unless its precedence bit is set (P), in which case it is executed immediately.

; **P,C** **[semi-colon]**

Terminates a colon definition and stops further compilation.

;S **P** **[semicolon-S]**

Stops interpretation of a screen from tape. It is also the word compiled by <;> at the end of a colon definition to return execution to the calling procedure.

< **(nl\n2 ... f)** **2 1** **[less-than]**

Leaves a true flag if nl is less than n2, otherwise leaves a false flag.

before: 15 2 17
after: 15 1 (true)

<# **[less-sharp]**

Sets up for numeric output formatting. The conversion is performed on a double number to produce text at PAD . See also # , #> , #S , SIGN .

<BUILDS **C** **[builds]**

Used within a colon definition in the form

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

to create a new defining word cccc

When cccc is executed in the form:

cccc nnnn

a new dictionary entry is created for nnnn with a name and a parameter area produced by <BUILDS and a high level execution procedure defined by DOES> .

When nnnn is executed it has the address of its parameter area (defined by <BUILDS) on the stack and executes the words after DOES> in cccc .

<CMOVE **(from\to\count ...)** **3 0** **[reverse-C-move]**

As CMOVE , but the source byte with highest address moves first and bytes are transferred in sequence of decreasing addresses. It is useful for short forward movements of memory contents, in cases where CMOVE would over-write the source data.

<TABLE! **(addr1\n ... addr2)** **2 1** **[reverse-table-store]**

Headerless code; execution address #3B3F. Places a value n at address addr1 and decrements the address by two to give addr2, ready for a new value. This can be used to store data in a table which is filled from the highest address towards lower addresses. It is used by LOAD to construct the address table for OSLOAD .

= **(n1\n2 ... f)** **2 1** **[equals]**

Leaves a true flag if n1 is equal to n2, otherwise leaves a false flag.

```
before:  53  27  27
after:   53  1
```

> **(n1\n2 ... f)** **2 1** **[greater than]**

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

```
before: 12 0 -1
after:  12  1
```

>R **(n ...)** **1 0 C** **[to-R]**

Removes a number from the computation stack and places it on the return stack. Its use must be balanced with R> in the same definition. It is used temporarily to remove a number from the stack to access a lower number. See R> .

? **(addr ...)** **1 0** **[question-mark]**

Prints the value contained in the two bytes starting at addr. Equivalent to <@ .> .

?COMP **[query-comp]**

Issues an error message if not compiling.

?CSP **[query-C-S-P]**

Issues an error message if stack position differs from that saved in CSP . Used as part of the compiler security.

?ERROR **(f\n ...)** **2 0** **[query-error]**

Issues error message number n if the boolean flag f is true. Uses ERROR . The stack is always empty after an error message.

?ESC **(... f)** **0 1** **[query-esc]**

Tests the keyboard to see if the ESC key is depressed. A true (non-zero) flag is returned if the key is down at the time of the test, otherwise a false (zero) flag is returned. In many FORTH systems this function is carried out by the word ?TERMINAL which may test for any key being pressed.

?EXEC **[query-exec]**

Issues an error message if not executing.

?LOADING **[query-loading]**

Issues an error message if not loading from tape.

?PAIRS **(n1\n2 ...)** **2 0** **[query-pairs]**

Issues an error message if n1 does not equal n2. The message indicates that compiled conditionals (IF ... ELSE ... THEN or BEGIN ... UNTIL etc.) do not match. It is part of the compiler security. The error message is given if, for example, the sequence IF ... UNTIL is found during compilation of a dictionary entry.

?STACK **[query-stack]**

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

?TERMINAL

See ?ESC .

@ **(addr ... n)** **1 1** **[fetch]**

Leaves on the stack the 16-bit value s found at addr.

```
before:  11  4123
after:   11   375
```

(assuming the value 375 was stored in the two bytes from address 4123).

ABORT

Clears both stacks, enters the execution state, prints the start-up message on the terminal device and returns control to the keyboard.

ABS **(n ... u)** **1 1**

Leaves u as the absolute value of n.

```
before: 12   -17
after:  12    17
```

AGAIN **P,C**

Used in a colon definition in the form

BEGIN ... AGAIN

During execution of a word containing this sequence, AGAIN forces a branch back to the corresponding BEGIN to create a endless loop.

ALLOT **(n ...)** **1 0**

The value of n is added to the dictionary pointer to reserve n bytes of dictionary space. The dictionary pointer may be moved backwards by use of a negative n but this should be used with caution to avoid losing essential dictionary content.

AND **(n1\n2 ... n3)** **2 1**

Leaves as n3 the bit-by-bit logical AND of n1 and n2.

(assuming binary)

before: 1101 1010 1100
after: 1101 1000

BACK **(addr ...)** **1 0**

Calculates the backward branch offset from HERE to addr and compile into the next available dictionary memory address. Used in the compilation of conditionals (AGAIN UNTIL etc.)

BASE **(... addr)** **0 1**

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

BEGIN **0 1 P,C**

Used in a colon definition in the forms:

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

BEGIN marks the start of a sequence that may be executed repeatedly. It acts as a return point from the corresponding AGAIN , UNTIL or REPEAT .

BL **(... c)** **0 1** **[B-L]**

A constant that leaves the ASCII value for 'blank' or 'space' (Hex 20).

BLANKS **(addr\n ...)** **2 0**

Fill n bytes of memory starting at addr with blanks.

BLK **(... addr)** **0 1** **[B-L-K]**

A user variable indicating the input source. If BLK is zero, input is taken from the keyboard. If it is non-zero input is taken from the tape input buffer area.

BRANCH

Headerless code; execution address #288D. The run-time procedure to cause an unconditional branch. The following in-line value is added to the interpretive pointer to cause a forward or backward branch. It is compiled by ELSE , AGAIN and REPEAT .

C! **(b\addr ...)** **2 0** **[C-store]**

Stores byte b (8 bits) at addr.

```
before: 53  29  3127
after:  53
```

(29 is stored in the single byte at address 3127)

C, **(b ...)** **1 0** **[C-comma]**

Stores (compiles) b in the next available dictionary byte, advancing the dictionary pointer by one.

C/L **(... n)** **0 1** **[C-slash-L]**

A constant containing the number of characters per line. This is normally 64, so a full FORTH 'line' will occupy two lines of the VDU display.

C@ **(addr ... b)** **1 1** **[C-fetch]**

Leaves b as the 8-bit contents of addr.

CFA **(pfa ... cfa)** **1 1** **[C-F-A]**

Converts the parameter field address of a word to its code field (execution) address.

CLIT **(... n)** **0 1** **[C-lit]**

Headerless code; execution address #285B. Within a colon-definition CLIT can be compiled before an 8-bit literal value. When the word containing CLIT is later executed the 8-bit value (in the range 0-255) is pushed to the stack as a single-precision (16-bit) number with its most-significant part set to zero.

This word is used by a number of system words but is not available to the user via the keyboard interpreter, which uses only LIT .

CMOVE **(from\to\count ...)** **3 0**

Moves 'count' bytes, starting at 'from' to the block of memory starting at 'to'. The byte at 'from' is moved first and the transfer proceeds towards high memory. No check is made as to whether the destination area overlaps the source area.

COLD

The cold start procedure used on first entry to the system. The dictionary pointer and user variables are initialised from the boot-up parameters and the system re-started via ABORT . It may be called from the keyboard to remove all application programs and restart with the nucleus dictionary alone.

COMPILE

COMPILE acts during the execution of the word containing it. The code field (execution) address of the word following COMPILE is compiled into the dictionary instead of executing, cf. [COMPILE] .

CONSTANT (n ...) 1 0

A defining word used in the form

n CONSTANT cccc

to create a word cccc , with the value n contained in its parameter field. When cccc is executed the value n will be left on the stack.

CONTEXT (... addr) 0 1

A user variable leaving the address of a pointer to the VOCABULARY in which a dictionary search will start.

COUNT (addr1 ... addr2\n) 1 2

Leaves the address addr2 and byte count n of a text string starting at addr1, in a form suitable for use by TYPE . It is assumed that the text string has its count byte at addr1 and that the actual character string starts at addr1 + 1.

(assuming that the text string 3 65 66 67 starts at 6124),

before: 47 6124
after: 47 6125 3

(TYPE would then display the 3 characters ABC)

CR [C-R]

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

CREATE

A defining word used in the form

CREATE cccc

to create a dictionary header for the word cccc with its code field pointing to the first byte of the parameter field.

One common use is, with the aid of <,>, to compile machine code into the parameter area to produce a machine code primitive. This does not need an assembler vocabulary.

CSP (... addr) 0 1 [C-S-P]

A user variable used for temporary storage of the stack pointer in checking of compilation errors.

CURRENT (... addr) 0 1

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+ (nd1\nd2 ... nd3) 4 2 [D-plus]

Leaves as nd3 the double number sum of double number nd1 and nd2.

D+- **(ndl\n ... nd2)** **3 2** **[D-plus-minus]**

Applies the sign of single number n to the double number ndl, leaving the result nd2. See +- .

D. **(nd ...)** **2 0** **[D-dot]**

Prints the signed double number nd according to the current BASE. One blank is printed after the number. See <.>.

D.R **(nd\n ...)** **3 0** **[D-dot-R]**

Prints a signed double number nd on the right of a field n characters wide. See .R . No trailing blank is printed.

DABS **(nd ... ud)** **2 2**

Leaves the absolute value ud of a signed double number nd. See ABS .

DECIMAL

Sets BASE to decimal numeric conversion for input and output.

DEFINITIONS

Sets the CURRENT vocabulary to the CONTEXT vocabulary. If used in the form

cccc DEFINITIONS

where cccc is a VOCABULARY word, all subsequent definitions will be placed in the vocabulary cccc .

DIGIT **(c\nl ... n2\tf)** **2 2**
(valid)
(c\nl ... ff) **2 1**
(invalid)

Converts ASCII character c, wih base nl, to its binary equivalent n2 and a true flag. If c is not a valid character in base nl, then only a false flag is left.

In hexadecimal base, but displaying stack in binary.

a) Valid
Hex character 'D' has ASCII code #44, or 01000100 in binary, and represents the value 1101 in binary

```
before: 00010110   01000100   00010000
after:  00010110   00001101   00000001
```

b) Invalid
Character 'G' has ASCII code #47, or 01000111 in binary, and does not represent a hexadecimal value

```
before: 00010110   01000111   00010000
after:  00010110   00000000
```

DLITERAL (nd ...) **2 0 P**
(compiling)

In the compiling state a double number nd is compiled as a double literal number in the dictionary. Later execution of the word including this literal number will replace nd on the stack.

In the execution mode DLITERAL has no effect.

DMINUS (nd1 ... nd2) **2 2**

Change the sign of nd1, leaving it as nd2.

DO (n1\n2 ...) **2 0 P,C**

May only be used within a colon definition in the forms

```
n1 n2 DO ... LOOP
n1 n2 DO ... +LOOP
```

This is the equivalent of a FOR ... NEXT loop in BASIC, repeating a sequence of operations a fixed number of times. The value of n1 is the loop limit and n2 is the initial value of the loop index. The loop terminates when the loop index equals or exceeds the limit. The sequence of operations in the loop will always be executed at least once. See I , LOOP , +LOOP , LEAVE .

DOES> **[does]**

Used with <BUILDS to create a new defining word cccc . When a word nnnn (created with cccc) executes it uses the sequence of operations following DOES> in cccc . At the start of this sequence the address of the parameter field of nnnn will be put on the stack so that the execution can refer to the particular values associated with nnnn . See <BUILDS .

DP (... addr) **0 1** **[D-P]**

A user variable, the dictionary pointer, leaves addr, whose contents point to the first free byte at the top of the dictionary.

DPL (... addr) **0 1** **[D-P-L]**

A user variable containing the number of digits to the right of the decimal point on double number input. It may also be used to contain the column location of a decimal point in user-generated output formatting. On single number entry the value in DPL defaults to -1.

DROP (n ...) **1 0**

Drops the top number on the stack.

```
before: 53  21
after:  53
```

DUP (n ... n\n) **1 2**

Duplicates the top number on the stack.

```
before: 53  21
after:  53  21  21
```

ELSE **P,C**

Used in a colon definition in the form

IF ... ELSE ... THEN

During execution ELSE causes a branch to the words after THEN if the flag tested by IF was true, and is the destination of the branch taken at IF if the flag was false. See IF .

EMIT (c ...) **1 0**

Transmits ASCII character c to the output device. The contents of OUT are incremented for each character output.

```
before:  23     65
after:   23
(A is displayed on the output).
```

ENCLOSE (addr\c ... addr\n1\n2\n3) 2 4

Headerless code; execution address #29B1. The text-scanning primitive used by WORD .

The text starting at addr is searched, ignoring leading occurrences of the delimiter c, until the first non-delimiter character is found. The offset from addr to this character is left as n1. The search continues from this point until the first delimiter after the text is found. The offsets from addr to this delimiter and to the first character not included in the scan are left as n2 and n3 respectively. The search will, regardless of the value of c, stop on encountering an ASCII null (00) which is regarded as an unconditional delimiter. The null is never included in the scan.

Examples:

Text at addr	n1	n2	n3
ccABCDcc	2	6	7
ABCDcc	0	4	5
ABC0cc	0	3	3
0ccc	0	1	0

ERASE (addr\n ...) **2 0**

Sets n bytes of memory starting at addr to contain zeros.

ERROR (n ...) **1 0**

Gives an error notification. The value in WARNING is examined and if it is -1 a system ABORT is executed, via (ABORT) which is a headerless dictionary entry. Otherwise an error message number n is given, the stacks are emptied and control is returned to the keyboard. In the system as supplied, WARNING is set to zero so the error message is given. Changing WARNING to -1 and also the vector in (ABORT) allows the user to define his own error response.

EXECUTE (addr ...) **1 0**

Executes the definition whose code field (execution) address is on the stack.

EXPECT **(addr\count ...)** **2 0**

Transfers characters from the keyboard to the memory starting at addr until a RETURN (#0D) is found or until the maximum count of characters has been received. Backspace deletes characters from both the display and the memory area but will not move past the starting point at addr. One or more nulls are added at the end of the text.

FENCE **(addr ...)** **1 0**

A user variable containing an address below which the user is not allows to FORGET . In order to use FORGET on an entry below this point it is necessary to alter the contents of FENCE . In the ATOM, changing the contents of FENCE to a value less than #8000 (the start of the upper block of RAM) may produce unpredictable results for FORGET .

FILL **(addr\n\b ...)** **3 0**

Fills n bytes of memory starting at addr with the value b.

FIRST **(... n)** **0 1**

A constant that leaves the address n of the first byte of the tape input/output buffer area.

FORGET **E**

Used in the form

FORGET cccc

to delete the definition with name cccc and all dictionary entries following it. An error message is given if the CURRENT vocabulary is not the same as the CONTEXT vocabulary, i.e. if the entry cccc is not in the vocabulary that is searched first. An error message is also given if cccc is in the protected area of the dictionary, below FENCE . Regardless of the value stored in FENCE the nucleus dictionary, all of which is necessary for the correct operation of the system, is protected against FORGET .

FORTH **P**

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. It is IMMEDIATE so it will execute if used during the creation of a colon definition. Until other vocabularies are defined, all new words become a part of FORTH . All other vocabularies ultimately link to the FORTH vocabulary.

HERE **(... addr)** **0 1**

Leaves the contents of DP i.e. the address of the first unused byte in the dictionary.

HEX

Sets the numeric conversion BASE to sixteen (hexadecimal).

HLD **(... addr)** **0 1** **[H-L-D]**

A user variable containing the address of the latest character of text produced during numeric output conversion (by #).

HOLD **(c ...)** **1 0**

Used between <# and #> to insert an ASCII character c into a converted numeric string. 2E (hex) HOLD will place a decimal point in the string.

I **(... n)** **0 1 C**

Used in a DO ... LOOP to place the current value of the loop index on the stack. It must be used at the same level of nesting as the DO ... LOOP , i.e. it will not operate correctly if included in a colon definition word beween DO and LOOP .

ID. **(addr ...)** **1 0** **[I-D-dot]**

Prints the name of a word from its name field address on the stack.

IF **(f ...)** **1 0 P,C**

Used in a colon definition in the forms

a) IF (true) ... THEN
b) IF (true) ... ELSE (false) ... THEN

If the flag f is true the sequence of words after IF is executed and execution is then transferred to the word immediately following THEN . If f is false execution transfers

a) to the word following THEN , or
b) to the sequence of words following ELSE and subsequently to the first word after THEN .

IMMEDIATE

Sets the precedence bit of the most recently defined word so that it will execute rather than being compiled during the compilation of a word definition. See [COMPILE] .

IN **(... addr)** **0 1**

A user variable containing the byte offset to the present position in the input buffer (terminal or tape) from where the next text will be accepted.

INTERPRET

The outer text interpreter which either executes or compiles a text sequence, depending on STATE , from the current input buffer (terminal or tape). If the word name cannot be found after a search of the CONTEXT and then the CURRENT vocabularies, it is converted to a number using the current base. If this conversion also fails an error message is given.

If a decimal point is found as part of a number the position of the decimal pointer will be stored in DPL and a double number will be left on the stack. The number itself will not contain any reference to the decimal point.

KEY **(... c)** **0 1**

Leaves the ASCII value of the next terminal key pressed.

LATEST **(... addr)** **0 1**

Leave the name field address of the most recently defined word in the CURRENT vocabulary.

LEAVE **C**

Forces the termination of a DO ... LOOP at the first following time that LOOP or +LOOP is reached. This is done by setting the loop limit equal to the current value of the loop index, which is not changed. Execution will continue normally until reaching LOOP or +LOOP .

LFA **(pfa ... lfa)** **1 1** **[L-F-A]**

Convert the parameter field address, pfa, to its link field address, lfa.

LIMIT **(... addr)** **0 1**

A constant leaving the address of the first byte after the highest memory available for the tape I/O buffer.

LIT **0 1 C**

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 following two bytes to be pushed onto the stack.

LITERAL **(n ...)** **P,C**

During compilation the stack value n is compiled into the dictionary entry as a 16-bit (single-precision) number.

A possible use is

: nnnn ... [calculate a value] LITERAL ... ;

Compilation is suspended (by [ ) for a value to be calculated and then resumed (by]) for LITERAL to compile the value into the definition of nnnn.

LOAD **(n ...)** **1 0**

Searches the tape file for screen n and, when found, loads it into the tape buffer. The PLAY TAPE response is as described in *LOAD, page 140 of "Atomic Theory and Practice", with the addition that the message '>n' is given, indicating that a search is being made for screen n. The contents of the buffer is then interpreted. Interpretation will end at the end of the screen or at ;S unless --> is found, in which case loading will continue with screen n+1.

LOOP **P,C**

Used in a colon definition in the form

DO ... LOOP

During execution LOOP controls branching back to the corresponding DO , dependent on the loop index and loop limit. The loop index is incremented by one and tested against the loop limit. Branching to DO continues until the index is equal to or greater than the limit when execution continues with the word following LOOP .

M* **(n1\n2 ... nd)** **2 2** **[M-times]**

Leaves as the double-precision number nd the signed product of the two signed single-precision numbers n1 and n2.

M/ **(nd\n1 ... n2\n3)** **3 2** **[M-divide]**

Leaves, as n2 and n3 respectively, the signed remainder and signed quotient from the division of the double number dividend nd by the single divisor n1. The sign of the remainder is that of the dividend.

M/MOD **(ud1\u2 ... u3\ud4)** **3 3** **[M-divide-mod]**

Leaves, as ud4 and u3 respectively, the double quotient and remainder from the division of the double dividend ud1 by the divisor u2. All are unsigned integers.

MAX **(n1\n2 ... max)** **2 1**

Leaves as max the larger of n1 and n2.

MESSAGE **(n ...)** **1 0**

Prints on the output device error message number n.

MIN **(n1\n2 ... min)** **2 1**

Leaves as min the smaller of n1 and n2.

MINUS **(n1 ... n2)** **1 1**

Changes the sign of n1 and leaves the result as n2. The sign is changed by forming the two's complement.

MOD **(n1\n2 ... mod)** **2 1**

Leaves as mod the remainder of n1/n2 with the sign of n1.

NFA **(pfa ... nfa)** **1 1** **[N-F-A]**

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

NUCTOP **(... addr)** **0 1 A**

Headerless code, execution address #2822. A constant containing the address of the top of the nucleus dictionary. Used by FORGET to prevent accidental forgetting of the nucleus dictionary, all of which is required for the correct operation of the system.

NUMBER **(addr ... nd)** **1 2**

Converts the character string starting at addr with a character count byte, to the signed double number nd using the current numeric base. If the string contains a decimal point its position will be given in DPL. If a valid numeric conversion is not possible an error message will be given. Used by INTERPRET .

NOOP **[no-op]**

A no-operation in FORTH. One possible use is to reserve address space in a colon-definition for later over-writing by the execution address of a subsequent definition.

OR **(n1\n2 ... or)** **2 1**

Leaves as or the bit-by-bit logical OR of n1 and n2.

OSDATA **(addr1\n ... addr2)** **0 1 A** **[O-S-data]**

Headerless code; execution address #3B4B. The implementation-dependent routine that creates the zero-page data required by the operating system load or save, with the exception of the pointer to the filename (see "Atomic Theory and Practice" p.191). The beginning of this data, left as addr, is at #62.

OSLOAD **(addr\f ...)** **2 0 A** **[O-S-load]**

Headerless code; execution address #3B24. The implementation-dependent machine code used by (LOAD) to make the appropriate call to the operating system load routines. The meanings of addr and f are as for (LOAD) .

OUT **(... addr)** **0 1**

A user variable containing a value that is incremented by EMIT . It may be examined and changed by the user to control display formats.

OVER **(n1\n2 ... n1\n2\n1)** **2 3**

Copies the second stack item over the top item.

Before: 15 23 17
after: 15 23 17 23

PAD **(... addr)** **0 1**

Leaves the address of the text output buffer. This is always 68 bytes above HERE . Numeric output characters are stored downwards from PAD , character text is stored upwards.

PFA **(nfa ... pfa)** **1 1** **[P-F-A]**

Converts the name field address, nfa, of a dictionary entry to its parameter field address, pfa.

QUERY

Inputs up to 80 characters terminated by RETURN (#0D) from the keyboard. The text is stored in the terminal input buffer whose address is given by TIB. The value of IN is set to zero (in preparation for interpretation by INTERPRET).

QUIT

Clears the return stack, stops compilation and returns control to the keyboard. No message is given.

R **(... n)** **0 1**

Copy the top of the return stack to the computation stack. The action is identical to that of I .

R0 **(... addr)** **0 1** **[R-zero]**

A silent user variable (no dictionary entry) containing the initial address of the top of the return stack. It may be given a header in the dictionary by:

```
8 USER R0
```

R# **(... addr)** **0 1** **[R-sharp]**

A user variable which contains the location of the editing cursor for the Editor.

R> **(... n)** **0 1** **[R-from]**

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

REPEAT **P,C**

Used in a colon definition in the form

BEGIN ... WHILE ... REPEAT

In execution REPEAT forces an unconditional branch back to BEGIN .

ROLL **(n ...)** **1 1**

Rotates the top nl stack items so that the nth item is moved to the top.

```
1 ROLL has no effect
2 ROLL is equivalent to SWAP
3 ROLL is equivalent to ROT
```

An error message is given if n is less than 1.

ROT **(n1\n2\n3 ... n2\n3\n1)** **3 3**

Rotates the top three items on the stack, bringing the third item to the top

```
before:  92  17  28  12
after:   92  28  12  17
```

RP! **[R-P-store]**

Initialises the return stack pointer.

RP@ **(... addr)** **0 1** **[R-P-fetch]**

Leaves the address of the return stack pointer. Note that this points one byte below the last return stack value.

S->D **(n ... nd)** **1 2** **[S-to-D]**

Leaves as nd the signed single-precision number n converted to the form of a signed double-precision number (with unchanged value).

S0 **(... addr)** **0 1** **[S-zero]**

A silent user variable (no dictionary entry) containing the address which marks the initial top of the computation stack. It may be given a header in the dictionary by:

```
6 USER S0
```

SCR **(... addr)** **0 1** **[S-C-R]**

A user variable containing the number of the most recently referenced source text screen.

SIGN **(n\nd ... nd)** **3 2**

Stores an ASCII '-' sign in the converted numeric output string at PAD if n is negative. The sign of n is usually that of the double number to be converted. Although n is discarded the double number nd is kept either for further conversion or to be dropped by #> . SIGN may only be used between <# and #> .

SMUDGE

TOGGLEs the 'smudge bit' in the name header of the most recently created definition in the CURRENT vocabulary. This switches between enabling and disabling the finding of the entry during a dictionary search.
The name field is smudged during the definition of a word to prevent the incomplete definition from being found, and then smudged again on completion.

SP! **0 0** **[S-P-store]**

Initialises the computation stack pointer.

SP@ **(... addr)** **0 1** **[S-P-fetch]**

Leaves the value of the stack pointer on the stack. The value corresponds to the state of the stack before the operation.

```
before:  1   2
(address 58  56   54)
after:   1   2    56
```

SPACE

Transmits an ASCII blank to the output device.

SPACES **(n ...)** **1 0**

Transmits n ASCII blanks to the output device.

STATE **(... addr)** **0 1**

A user variable indicating the state of compilation. A zero value indicates execution and a non-zero value indicates compilation.

SWAP **(n1\n2 ... n2\n1)** **2 2**

Exchanges the top two items on the stack.

```
before: 17  23  59
after:  17  59  23
```

THEN **P,C**

Used in a colon definition in the forms

```
IF ... THEN
IF ... ELSE ... THEN
```

Marks the destination of forward branches from IF or ELSE as the conclusion of the conditional structure . See IF .

TIB **(... addr)** **0 1**

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

TRAVERSE (addr1\n ... addr2) 2 1

Headerless code; execution address #2FB8. Moves across the name field of a dictionary entry. If n=1, addr1 should be the address of the name length byte (i.e. the NFA of the word) and the movement is towards high memory. If n=-1, addr1 should be the last letter of the name and the movement is towards low memory. The addr2 that is left is the address of the other end of the name.

TOGGLE (addr\b ...) 2 0

Complements the contents of addr by the bit pattern b.

```
before: b = 00110000, contents of addr = 01101010
after:                contents of addr = 01011010
```

TYPE (addr\count ...) 2 0

Transmits count characters of a string starting at addr to the output device.

U* (u1\u2 ... ud) 2 2 [U-times]

Leaves the unsigned double-precision product of two unsigned numbers.

U. (n ...) 1 0 [U-dot]

Transmits the 16-bit value n to the output device. n is represented as an unsigned integer in the current numeric conversion base. A trailing space is printed.

U/ (ud\u1 ... u2\u3) 3 2 [U-divide]

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

U< (un1\un2 ... f) 2 1

Unsigned comparision. Leaves a true flag if un1 is less than un2, otherwise leave a false flag. For correct operation the difference between un1 and un2 should not exceed 32767.

UNTIL (f ...) 1 0 P,C

Used in a colon definition in the form

BEGIN ... UNTIL

If f is false execution branches back to the corresponding BEGIN . If f is true execution continues with the next word after UNTIL .

USER (n ...) 1 0

A defining word used in the form

n USER cccc

to create a user variable cccc . Execution of cccc leaves the address, in the user area, of the value of cccc . The value of n is the offset from the start of the user variable area to the memory location (2 bytes) in which the value is stored. The value is not initialised.

VARIABLE (n ...) 1 0

A defining word used in the form

n VARIABLE cccc

to create a variable cccc with initial value n. Execution of cccc leaves the address, in the parameter area of cccc , containing the value of cccc .

VLIST [V-list]

Display, on the output device, a list of the names of all words in the CONTEXT vocabulary and any other vocabulary from which the CONTEXT vocabulary is chained. All VLIST's will therefore include a listing of words in the FORTH vocabulary. The listing be be interrupted by pressing the ESC key and resumed by pressing the space bar. If, after interruption, The ESC key (or any other key except the space bar) is pressed, the listing will be aborted.

VOC-LINK (... addr) 0 1

A user variable containing the address of a vocabulary link field in the word which defines the most recently created vocabulary. All vocabularies are linked through these fields in their defining words.

VOCABULARY E

A defining word used in the form

VOCABULARY cccc

to create a defining word for a vocabulary with name cccc . Execution of cccc makes it the CONTEXT vocabulary in which a dictionary search will start. Execution of the sequence:

cccc DEFINITIONS

will make cccc the CURRENT vocabulary into which new definitions are placed. Vocabulary cccc is so linked that a dictionary search will also find all words in the vocabulary in which cccc was originally defined. All vocabularies, therefore, ultimately link to FORTH .

By convention all vocabulary defining words are declared IMMEDIATE .

WARNING (... addr) 0 1

A user variable whose value determines the action on detection of an error. If WARNING contains -1 an error causes a system ABORT which may, by changing a pointer in (ABORT) , be altered to a user-defined response. If WARNING contains zero then an error message number is given. In the system provided, WARNING is set to zero on initialisation . See ERROR .

WHILE (f ...) 1 0 P,C

Used in a colon definition in the form

BEGIN ... WHILE ... REPEAT

WHILE tests the top value on the stack. If it is true execution continues to REPEAT which forces a branch back to BEGIN . If f is false execution skips to the first word after REPEAT . See BEGIN .

WIDTH **(... addr)** **0 1**

A user variable containing the maximum number of letters saved during the compilation of a definition's name. It must be a value between 1 and 31 inclusive and has a default value of 31. The value may be changed at any time provided it is kept within the above limits.

WORD **(c ...)** **1 0**

Accepts text characters from the input buffer (terminal or tape) until a delimiter character c is found. The string starting with a length count byte, is then placed in the WORD buffer starting at HERE and two or more banks are added to the end. The choice of input buffer is determined by BLK . See BLK , IN .

X

This is a pseudonym for the dictionary entry whose name is one character of ASCII null (00). It is the procedure to terminate interpretation of text from the input buffer, since both input buffers have at least one null character at the end.

XOR **(n1\n2 ... xor)** **2 1**

Leaves the bit-by-bit logical exclusive-OR of n1 and n2.

[**P** **[left-bracket]**

Used in the creation of a colon definition in the form

: nnnn ... [...] ... ;

to suspend compilation of the definition and allow words to execute . See] .

[COMPILE] **P,C** **[bracket-compile]**

Used in the creation of a colon definition to force the compilation of an IMMEDIATE word which would otherwise execute.

The most frequent use is with vocabulary words e.g.

[COMPILE] FORTH

to delay the change of the CONTEXT vocabulary to FORTH until the word containing the above sequence executes.

] **[right bracket]**

Used during the creation of a colon definition, to resume compilation after the suspension of compilation by [.

Appendix A
Two's-Complement Arithmetic

In unsigned arithmetic using 16-bit numbers, the lowest value that can be represented is zero, appearing as binary notation as

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ,

and the highest number appears as

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

which represents the decimal value 65535. There are therefore, including zero, 65536 different numbers.

To understand the operations on signed numbers, consider what happens if one is added to the highest unsigned value, 65535. In binary notation this sum appears as

```
          1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    +     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
    ---------------------------------------
(1)       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    ---------------------------------------
```

In a computer, working to 16-bit accuracy, the one in the 17th place is lost and the value stored as the result will be zero. If we add one to a number and find the result is zero, it is natural to interpret the original number as having a value of -1.

Thus, for signed arithmetic, the number

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

can be used to represent -1.

In general the number -x is represented by the value which gives a zero result when +x is added to it (ignoring any overflow into the 17th place). The signed values -2 , -23 and -32768 are therefore represented by

```
          1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
          1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1
and       1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0     respectively.
```

All negative values are represented by binary numbers whose most significant (16th) bit is a one. Accordingly, the highest positive number that can be represented is:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

or +32767, and the most negative number is -32768, shown earlier.

The range for a signed number is thus from -32768 to +32767 which, including zero, gives a total of 65536 different values (as for unsigned numbers).

Whether a number is interpreted as a signed or an unsigned value is entirely a matter of context; the binary number

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

may represent either +65535 or -1 depending on the conversion routine used.

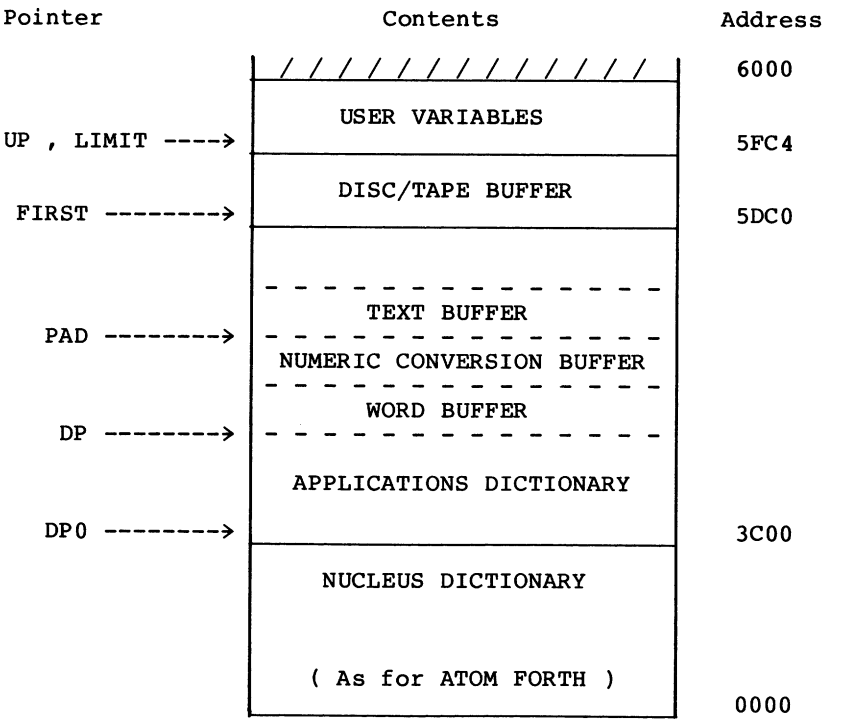
The above discussion has been confined to 16-bit numbers but similar considerations apply to any precision. In all cases the most significant bit of the number will be zero for a positive value and one for a negative value. It may, therefore, be regarded as a 'sign bit'.

In general the binary representation of a negative number may be found by writing down the binary representation of the corresponding positive number, inverting all the bits and adding one. This is shown in the following example to find the two's-complement representation of -4 (in 8-bit precision):

```
            0 0 0 0 0 1 0 0    (+4)

invert all bits (form the one's-complement):

            1 1 1 1 1 0 1 1

add 1 (form the two's-complement):

            1 1 1 1 1 1 0 0    (-4)
```

Appendix B
System FORTH

FORTH for the Acorn Systems 3 and 4 is almost identical to ATOM FORTH, except for the changes in the memory map. FORGET is changed to work in the range #3C00 and #8000 and no memory is needed above #8000.



The system can be modified to use memory up to #8000 by changing UP, LIMIT and FIRST, for example:

```
HEX
LIMIT 2000 + ' LIMIT !
FIRST 2000 + ' FIRST !
10 +ORIGIN @ 2000 + 10 +ORIGIN !          ( change UP )
COLD                                      ( initialise )
```

The system should then be resaved:

```
SAVE FORTHA 240 400 2EB
SAVE FORTHB 2800 3C00 2EB
```

Appendix C
Dictionary Entry Structure

All dictionary entries in FORTH have the same general form:

NFA	Name length (1 byte), Characters of the name (up to 31 bytes)	Name field
LFA	Link pointer to previous NFA (2 bytes)	Link field
CFA	Pointer to machine code to execute (2 bytes)	Code field
PFA		Parameter field

The name length byte contains, in its least significant five bits, the number of characters in the name of the word (maximum 31 characters). The sixth bit is the 'smudge' bit which, when set to 1, will prevent the dictionary entry from being found on a dictionary search (except by VLIST). This is mainly used to prevent the finding and use of a partly-completed dictionary entry. The seventh bit is known as the precedence bit and marks a word as being IMMEDIATE when set to 1. The eighth or most significant (sign) bit is always set to 1, as is the sign bit of the last character of the name. This is to allow the operation of TRAVERSE , which will move, in either direction, across the name field of the word.

The link field contains the address of the start of the name field of the preceding dictionary entry to allow a dictionary search to be made. A link field containing zero marks the end of the dictionary.

The various types of dictionary entry differ only in the contents of their code fields and parameter fields. The code field always contains a pointer to the start of an executable machine code routine, and the different possibilities are given in the following list.

a) Machine-code primitives

   The machine code is placed in the parameter field of the entry and the code field points to its start.

b) Constants

   The value of the constant is contained in a two-byte parameter field and the code field points to a machine-code routine to place the contents of the parameter field on the stack.

c) Variables

   The value of the variable is contained in a two-byte parameter field and the code field contains a pointer to a machine-code routine which places the address of the parameter area on the stack.

d) User Variables

The offset from the start of the user variable area to the address where the variable is stored is contained in a one-byte parameter area. The code field points to a machine-code routine which adds the offset to the address of the start of the user variable area and places the result on the stack.

e) Colon definitions

The code field contains a pointer to a machine-code routine which interprets the contents of the parameter field as a list of addresses of other FORTH words to be executed.

f) Words constructed using <BUILDS and DOES>

The first two bytes of the parameter area contain the address of the words following DOES> in the creating word. The remainder of the parameter area contains a series of values placed there by the words (if any) following <BUILDS in the creating word.

The code field contains a pointer to machine code which will

(i) place the address of the third byte of the parameter area (the start of the values placed there by <BUILDS) on the stack;

(ii) execute the list of words starting at the address contained in the first two bytes of the parameter area (the words following DOES>).

Saving Dictionary Space

The maximum length of the name of a dictionary entry is contained in the user variable WIDTH . This may at any time be reduced from its default value of 31 characters, with a consequent saving of space in the name field of a dictionary entry. If, for example, the value of WIDTH is reduced to 3 by

```
3 WIDTH !
```

then any new dictionary entry will be with the actual length of its name, but only the first three characters saved. All words must then be uniquely determined by their length and their first three characters (i.e. LOOK and LOOP will not be distinguished). The use of a value of WIDTH less than 3 is not recommended but, as this demonstrates,

```
IT IS VER- EAS- TO REA- FOR-- IF YOU ONL- HAVE- THE FIR-- THR--
LET---- AND THE LEN--- OF THE WOR-
```

The value of WIDTH may be increased or decreased at any time (subject to its remaining in the range 3 to 31 inclusive) and will not have any effect on previously defined words.

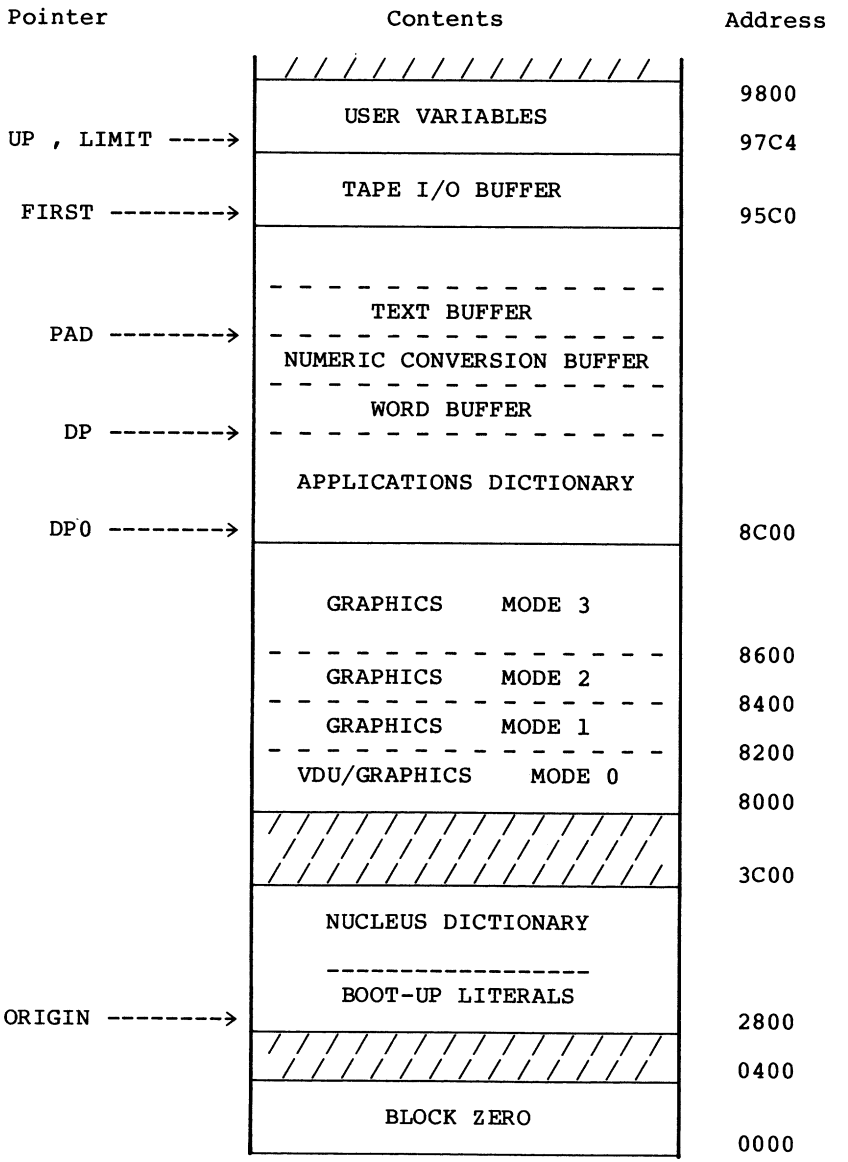
Headerless Code

Some dictionary entries in ATOM FORTH are 'headerless'. This means that their heads do not include name and link fields. They cannot, therefore, be found by a dictionary search, or included in a new definition unless their code field (execution) addresses are known to the programmer. The glossary gives this address for each headerless entry.

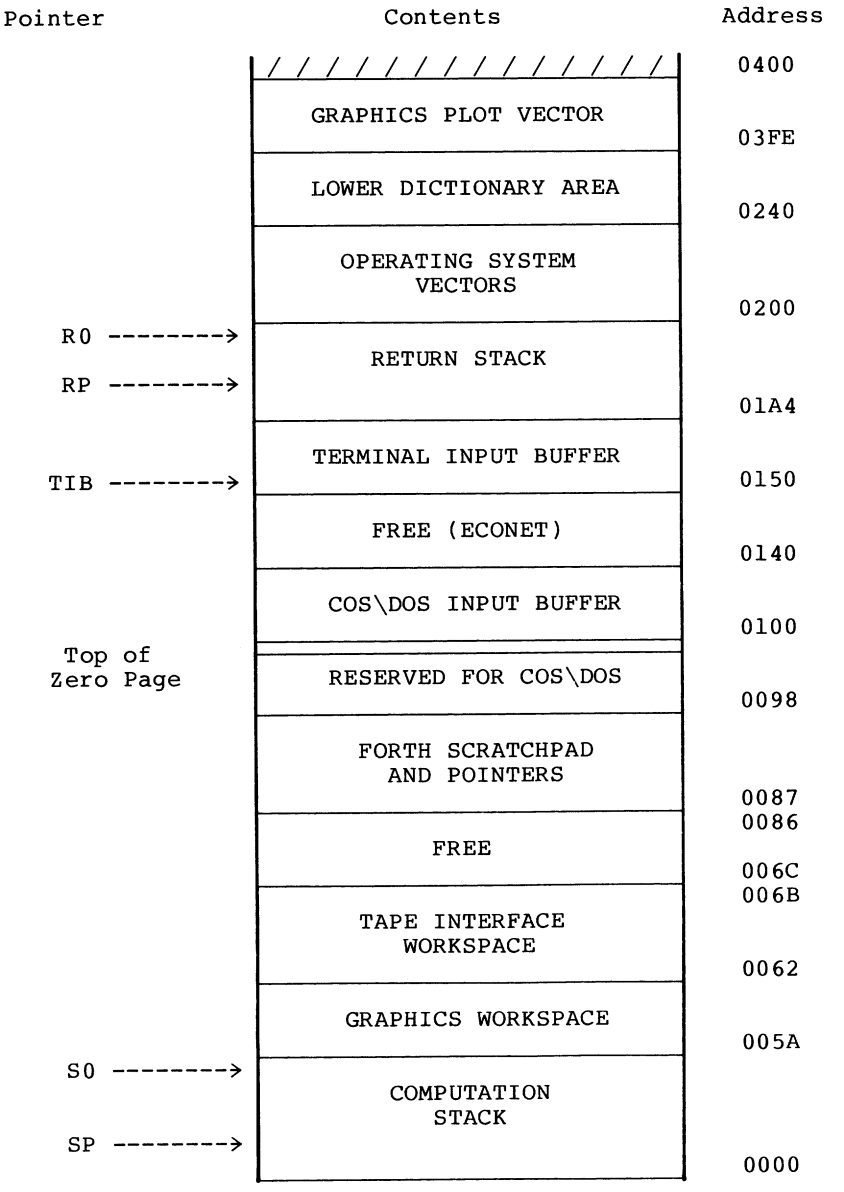
Creating headerless code is a very efficient way of saving memory, but it does reduce the flexibility of the system since headerless entries are relatively difficult to use.

The main use of headerless code is in producing a stand-alone system whose action is fixed, such as a dedicated control system.

Appendix D
Memory Allocation



Block zero Memory Map



Relocation of the Applications Dictionary

The applications dictionary, tape buffer and user variable area normally reside in memory between #8200 and #97FF. All addresses in this range are, if treated as signed single-precision integers, 'negative'. This has two important consequences for relocation of the applications dictionary in the address range below #8000:

1. Before compiling a dictionary entry, FORTH checks whether there is sufficient room between the top of the dictionary (contents of DP) and the start of the tape buffer (value of FIRST). A 'positive' value in DP and a 'negative' value of FIRST will cause the test to fail and give error message 2 (dictionary full). In order to avoid extensive changes to the system, relocation of the applications dictionary below #8000 should be accompanied by a corresponding relocation of the tape buffer.

2. The word FORGET performs an address validation check before allowing part of the dictionary to be discarded. This check assumes that valid addresses are 'negative', and they will therefore fail in the relocated applications dictionary. The definition of FORGET must be modified for the relocated system.

The modifications given here also relocate the user variables. This is not strictly necessary but will leave the upper RAM completely free so that mode 4 graphics can be used. The changes are made permanent so that the new system can be saved on tape. The areas to be saved are #240 to #400 and #2800 to #3C00, both with an execution address of #2800. A minimum of 2K of RAM is required in the region between #3C00 and #7FFF inclusive. In the code, XXXX represents the RAM start address and YYYY the end address + 1.

```
COLD                        ( start on an empty applications dictionary )
HEX

XXXX DUP DP !               ( relocate applications dictionary )
DUP 1E +ORIGIN !            ( change DP boot-up parameter )
DUP 1C +ORIGIN ! FENCE !    ( FENCE and its boot-up parameter )

YYYY 3C -                   ( start of new user variable area )
DUP 10 +ORIGIN !            ( change user variable boot-up parameter )

DUP ' LIMIT !               ( relocate end of tape buffer )
204 - ' FIRST !             ( and the beginning )

: TEMP                      ( a temporary definition of the new FORGET )
    CURRENT @ CONTEXT @
    - 18 ?ERROR             ( CONTEXT = CURRENT? )
    [COMPILE] '
    DUP FENCE @
    [ 2822 , ]              ( top of nucleus )
    MAX < 15 ?ERROR         ( below FENCE or in nucleus? )
    DUP NFA DP !
    LFA @ CURRENT @ ! ;
```

If this definition is compiled successfully, all is well so far.

```
' TEMP HERE OVER -          ( PFA and parameter field length )
' FORGET SWAP CMOVE         ( overwrite FORGET )
                            ( the new version is shorter, so OK )
```

```
FORGET TEMP
                ( If TEMP is forgotten successfully, everything is OK )
COLD            ( Finally, check out modified boot-up parameters )
```

Appendix E
Further Reading

1. The FORTH Interest Group in the U.S.A. supply many documents relating to FORTH, including assembly listings for many different microprocessors, a language model, reprints of "BYTE" magazine articles and a bi-monthly magazine entitled "FORTH Dimensions".

For details of current costs for membership and their publications write (with an SAE please) to:

FORTH Interest Group,
P.O. Box 1105,
San Carlos,
Ca. 94070.

2. The FORTH Interest Group U.K. is the British branch of the U.S.A. group. At present it meets on the first Thursday of every month at 7 p.m. at the Polytechnic of the South Bank in London. Membership includes a bi-monthly newsletter entitled "FORTHWRITE". Like its parent group, F.I.G. U.K. exists to promote interest in and the use of the FORTH language and its members are prepared to help with any difficulties that may be encountered. For further details contact (S.A.E. please):

The Honorary Secretary,
F.I.G. U.K.,
15, St. Albans Mansions,
Kensington Court Palace,
London W8 5QH

3. A number of articles on FORTH have appeared in "BYTE" magazine:

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

The August 1980 issue is now unobtainable, but reprints of all the BYTE articles are available from the U.S.A. FORTH Interest Group (Ref. 1).

4. "FORTH for Microcomputers" Dr. Dobb's Journal No. 25 (May 1978). A brief review of the external and internal workings, with a variety of examples.

5. "Starting FORTH" L. Brodie.
published by Prentice-Hall (Nov. 1981).

Available from: Computer Solutions Ltd.,
Treway House,
Hanworth Lane,
Chertsey.
Tel: Chertsey (09328) 65292

The author is from FORTH Inc., the company started by Charles Moore, the inventor of FORTH. This is a very good introduction to the language, with lots of examples.

6. "Threaded Interpretive Languages" R. G. Loeliger,
published by Byte Books (McGraw-Hill) (1981).

A good, clear, description of the internal workings of FORTH-like languages, based on an implementation for the Z-80 microprocessor. Not for the complete beginner, but try it in a couple of month's time!

Index

! (FORTH word) 21, 92
!CSP (FORTH word) 92

" character 96

£ (FORTH word) 53, 92
£> (FORTH word) 53, 92
£S (FORTH word) 53, 92

$! examp_e 74
$@ example 74
$IN example 51, 73, 74

' (FORTH word) 92

((FORTH word) 60, 93
(+LOOP) (FORTH word) 93
(.") (FORTH word) 93
(;CODE) (FORTH word) 93
(ABORT) (FORTH word) 46, 93
(DO) (FORTH word) 93
(ENTER) (FORTH word) 93
(FIND) (FORTH word) 93
(LINE) (graphics word) 67
(LOAD) (FORTH word) 94
(LOOP) (FORTH word) 94
(NUMBER) (FORTH word) 46, 47, 94
(PLOT) (graphics word) 66

) character 60, 93

* (FORTH word) 12, 94
*/ (FORTH word) 12, 94
*/MOD (FORTH word) 12, 94

+ (FORTH word) 12, 94
+! (FORTH word) 22, 95
+- (FORTH word) 12, 95
+LOOP (FORTH word) 39, 95
+ORIGIN (FORTH word) 95

, (FORTH word) 24, 95

- (FORTH word) 12, 95
--> (FORTH word) 60, 95
-DUP (FORTH word) 13, 37, 95
-FIND (FORTH word) 95
-TRAILING (FORTH word) 45, 51, 96

. (FORTH word) 11, 45, 51, 96
." (FORTH word) 45, 96
.POUNDS example 53
.R (FORTH word) 51, 96
.REAL example 54
.S example 88
.TIME example 84

/ (FORTH word) 12, 96
/MOD (FORTH word) 12, 96

0 (FORTH word) 96
0< (FORTH word) 15, 96
0= (FORTH word) 15, 96
0BRANCH (FORTH word) 96

1 (FORTH word) 96
1+ (FORTH word) 12, 96

2 (FORTH word) 96
2* (FORTH word) 12, 97
2+ (FORTH word) 12, 97
2/ example 77
2ARRAY example 72
2DROP (FORTH word) 17, 97
2DUP (FORTH word) 17, 97

3-COUNT example 39

: (FORTH word) 20, 97

; (FORTH word) 20, 97
;S (FORTH word) 61, 97

< (FORTH word) 15, 97
<£ (FORTH word) 53, 97
<BUILDS (FORTH word) 69, 70, 97
<CMOVE (FORTH word) 48, 98
<TABLE! (FORTH word) 98

= (FORTH word) 15, 98

> (FORTH word) 15, 98
>R (FORTH word) 13, 98

? (FORTH word) 22, 98
?COMP (FORTH word) 98
?CSP (FORTH word) 98
?ERROR (FORTH word) 98
?ESC (FORTH word) 42, 99
?EXEC (FORTH word) 99
?LOADING (FORTH word) 99
?PAIRS (FORTH word) 99
?STACK (FORTH word) 99
?TERMINAL (FORTH word) 99

@ (FORTH word) 22, 99

ABORT (FORTH word) 99

ABS (FORTH word) 12, 99
AGAIN (FORTH word) 42, 99
allocation memory 125
ALLOT (FORTH word) 70, 100
AND (FORTH word) 15, 100
animated graphics 66
applications separating 21
arithmetic 10
  double-precision 16
  single-precision 11
  two's-complement 117
arrays 71
  one-dimensional 71
  two-dimensional 72
  with index check 71

B (editor command) 63
BACK (FORTH word) 100
BACKWARDS example 39
BASE (FORTH word) 23, 49, 100
base BASE-36 50
  BINARY 50
  conversion example 50
BASE-36 base 50
bases numeric 49
BEGIN (FORTH word) 42, 100
BINARY base 50
BL (FORTH word) 49, 100
BLACK (graphics word) 66
BLANKS (FORTH word) 49, 100
BLK (FORTH word) 23, 100
blocks of memory manipulating 48
BRANCH (FORTH word) 100
branches 35
  conditional 35
  nested 37
BREAK key 7

C (editor command) 62
C! (FORTH word) 101
C, (FORTH word) 24, 101
C/L (FORTH word) 73, 101
C@ (FORTH word) 101
CASE example 74, 75
CFA (FORTH word) 20, 101
character input 45
  output 50
CLEAR (FORTH word) 65
clear graphics 67
  screen 67
CLIT (FORTH word) 101
CMOVE (FORTH word) 48, 101
  intelligent 49
code field address 19
code headerless 123
coding example 4
COLD (FORTH word) 8, 101
cold start 8
colon-definitions 20
  compilation of 30
  form of 20
comments 60
compilation of colon-definitions 30
  IMMEDIATE words 31
  numbers 32
COMPILE (FORTH word) 31, 101
  example 32
compiler security 31
computation stack 10
conditional branches 35
CONSTANT (FORTH word) 21, 70, 102
CONTEXT (FORTH word) 23, 28, 102
conversion numeric 47
copying screens 79
COUNT (FORTH word) 45, 51, 102
COUNTER example 38
COUNTS example 38
cover figure 86
CR (FORTH word) 102
CREATE (FORTH word) 24, 102
  + example 27
  AND example 27
  RP@ example 28
CSP (FORTH word) 23, 102
CTABLE example 73
CURRENT (FORTH word) 23, 28, 102
cursor editing 62
CVARIABLE example 71, 71

D (editor command) 61
D+ (FORTH word) 16, 102
D+- (FORTH word) 17, 103
D->H example 50
D. (FORTH word) 51, 103
D.R (FORTH word) 52, 103
DABS (FORTH word) 17, 103
DATA example 73
DECIMAL (FORTH word) 50, 103
decimal point 16
definite loops 37
definitions 19
DEFINITIONS (FORTH word) 29, 103
DELAYS example 38
DELETE (editor command) 63
deleting lines 61
DELTAX (graphics word) 67
DELTAY (graphics word) 67
demonstration graphics 86
DIAGONAL example 66
dictionary entries 19, 121
  entry types 121
  relocation 127
dictionary space extra 65
  saving 122
DIGIT (FORTH word) 103
discs 64
DLITERAL (FORTH word) 32, 104
DMINUS (FORTH word) 17, 104
DO (FORTH word) 37, 104

DO-IT-LATER example 31
DO-IT-NOW example 30
double-precision arithmetic 16
  numbers 16
  operators 16
DP (FORTH word) 23, 104
DPL (FORTH word) 16, 23, 46, 47, 104
DROP (FORTH word) 13, 104
DUMP example 52
DUP (FORTH word) 13, 104

E (editor command) 61
econet 64
editing cursor 62
  example 58, 59, 63
  lines 61
  strings 62
editor 58
editor commands:
  B 63
  C 62
  D 61
  DELETE 63
  E 61
  F 62
  H 61
  I 61
  M 63
  MATCH 63
  N 63
  P 60, 61
  R 61
  S 61
  T 61
  TILL 62
  WHERE 57
  X 62
editor loading 55
EDITOR vocabulary 58
ELSE (FORTH word) 105
EMIT (FORTH word) 50, 105
EMPTY-BUFFERS (FORTH word) 58
ENCLOSE (FORTH word) 105
ERASE (FORTH word) 49, 105
erasing lines 61
ERR (graphics word) 67
ERROR (FORTH word) 105
error message from arrays 72
  messages 91
errors 91
examples:
  $! 74
  $@ 74
  $IN 51, 73, 74
  .POUNDS 53
  .REAL 54
  .S 88
  .TIME 84
  2/ 77

2ARRAY 72
3-COUNT 39
BACKWARDS 39
base conversion 50
CASE 74, 75
coding 4
COMPILE 32
COUNTER 38
COUNTS 38
CREATE + 27
CREATE AND 27
CREATE RP@ 28
CTABLE 73
CVARIABLE 71, 71
D->H 50
DATA 73
DELAYS 38
DIAGONAL 66
DO-IT-LATER 31
DO-IT-NOW 30
DUMP 52
editing 58, 59, 63
FACT 77, 78
factorials 77
FAMILY 69
fast 2 / 77
FLASH 66
GCD 43
HELL-FREEZES-OVER 43
IMMEDIATE 30, 32
INPUT 47
INSCR 80
INVERT editing 62
JTEST 41
KEYBOARD 83
LEFT$ 51
LIST 73
LOOK-UP 40, 41
MD* 17
MEMBER 69
MONITOR 81
NUMIN 46
OUTSCR 80
PAUSE 42
PLOT-IT 86
quadratic 12, 14
QUICKSORT 78
RECTANGLE 72
recursion 77
RIGHT$ 51
RND 59
SEQUENCE 39
SHOWASCII 45
SIZE 70
sort 78
STARS 7
STRING 73
STRINGS 51
TENCOUNT 37
tones 82

TRIANGLE 67
VALUES 71
WAIT 81
WASHING 4
WORDS 73
[COMPILE] 31, 32
EXECUTE (FORTH word) 105
execution address 19
execution time of words 85
EXPECT (FORTH word) 106
extension memory 67, 127
extra dictionary space 65

F (editor command) 62
FACT example 77, 78
factorials example 77
FAMILY example 69
fast 2 / example 77
FENCE (FORTH word) 23, 106
FILL (FORTH word) 49, 106
FIRST (FORTH word) 106
FLASH example 66
FORGET (FORTH word) 21, 28, 106
form of colon-definitions 20
FORTH (FORTH word) 29, 106
FORTH words:
! 21, 92
!CSP 92
£ 53, 92
£> 53, 92
£S 53, 92
' 92
(60, 93
(+LOOP) 93
(.") 93
(;CODE) 93
(ABORT) 46, 93
(DO) 93
(ENTER) 93
(FIND) 93
(LOAD) 94
(LOOP) 94
(NUMBER) 46, 47, 94
* 12, 94
*/ 12, 94
*/MOD 12, 94
+ 12, 94
+! 22, 95
+- 12, 95
+LOOP 39, 95
+ORIGIN 95
, 24, 95
- 12, 95
--> 60, 95
-DUP 13, 37, 95
-FIND 95
-TRAILING 45, 51, 96
. 11, 45, 51, 96
." 45, 96
.R 51, 96
/ 12, 96
/MOD 12, 96
0 96
0< 15, 96
0= 15, 96
0BRANCH 96
1 96
1+ 12, 96
2 96
2* 12, 97
2+ 12, 97
2DROP 17, 97
2DUP 17, 97
: 20, 97
; 20, 97
;S 61, 97
< 15, 97
<£ 53, 97
<BUILDS 69, 70, 97
<CMOVE 48, 98
<TABLE! 98
= 15, 98
> 15, 98
>R 13, 98
? 22, 98
?COMP 98
?CSP 98
? 98
?ESC 42, 99
?EXEC 99
?LOADING 99
?PAIRS 99
?STACK 99
?TERMINAL 99
@ 22, 99
ABORT 99
ABS 12, 99
AGAIN 42, 99
ALLOT 70, 100
AND 15, 100
BACK 100
BASE 23, 49, 100
BEGIN 42, 100
BL 49, 100
BLANKS 49, 100
BLK 23, 100
BRANCH 100
C! 101
C, 24, 101
C/L 73, 101
C@ 101
CFA 20, 101
CLEAR 65
CLIT 101
CMOVE 48, 101
COLD 8, 101
COMPILE 31, 101
CONSTANT 21, 70, 102
CONTEXT 23, 28, 102
COUNT 45, 51, 102

CR 102
CREATE 24, 102
CURRENT 23, 28, 102
D+ 16, 102
D+- 17, 103
D. 51, 103
D.R 52, 103
DABS 17, 103
DECIMAL 50, 103
DEFINITIONS 29, 103
DIGIT 103
DLITERAL 32, 104
DMINUS 17, 104
DO 37, 104
DOES> 69, 70, 104
DP 23, 104
DPL 16, 23, 46, 47, 104
DROP 13, 104
DUP 13, 104
editor commands 62
ELSE 105
EMIT 50, 105
EMPTY-BUFFERS 58
ENCLOSE 105
ERASE 49, 105
ERROR 105
EXECUTE 105
execution time of 85
EXPECT 106
FENCE 23, 106
FILL 49, 106
FIRST 106
FORGET 21, 28, 106
FORTH 29, 106
HERE 45, 51, 106
HEX 50, 106
HLD 23, 106
HOLD 53, 107
I 38, 107
ID. 107
IF 35, 107
IMMEDIATE 30, 107
IN 23, 107
in text 1
INTERPRET 107
J 41
K 42
KEY 107
LATEST 77, 107
LEAVE 40, 108
LFA 20, 108
LIMIT 108
LIST 56
LIT 108
LITERAL 32, 108
LOAD 56, 108
LOOP 37, 108
M* 16, 108
M/ 17, 109
M/MOD 17, 109
MAX 15, 109
MESSAGE 109
MIN 15, 109
MINUS 12, 109
MOD 12, 109
names of 89, 122
NFA 20, 109
NOOP 109
NUCTOP 109
NUMBER 46, 109
OR 15, 109
OSDATA 110
OSLOAD 110
OUT 23, 110
OVER 13, 110
PAD 54, 61, 110
PFA 20, 110
PICK 13
PROGRAM 58
pronunciation of 89
QUERY 45, 110
QUIT 110
R 13, 110
R£ 23, 111
R0 23, 110
R> 13, 111
REPEAT 42, 43, 111
ROLL 13, 111
ROT 13, 111
RP! 111
RP@ 111
S->D 111
S0 23, 111
SCR 23, 57, 111
SIGN 53, 112
SMUDGE 16, 24, 112
SP! 112
SP@ 112
SPACE 112
SPACES 112
stack actions of 89
STATE 23, 112
status of 89
SWAP 13, 112
THEN 35, 112
TIB 23, 112
TOGGLE 15, 113
TRAVERSE 113
TYPE 45, 50, 51, 113
U* 16, 113
U. 11, 52, 113
U/ 17, 113
U< 15, 113
UNTIL 42, 113
USER 22, 113
validity of 4
VARIABLE 22, 114
VLIST 7, 114
VOC-LINK 23, 114
VOCABULARY 19, 28, 114

WARM 8
WARNING 23, 46, 114
WIDTH 23, 115, 122
WORD 45, 115
X 115
XOR 15, 115
[30, 115
[COMPILE] 30, 115
] 30, 115
further reading 129

GCD example 43
glossary of FORTH words 89
GOTO 35
graphics 65, 86
animated 66
clear 67
demonstration 86
line-drawing 66
memory 65
mode 4 67
modes 65
package 65
point-plotting 66
relative plotting 67
resolution 65
GRAPHICS vocabulary 66
graphics words:
(LINE) 67
(PLOT) 66
BLACK 66
DELTAX 67
DELTAY 67
ERR 67
INVERT 66
LINE 67
MOVE 67
PLOT 66
REL 67
RLINE 67
RMOVE 67
RPLOT 67
SETXY 67
WHITE 66
XDIR 67
YDIR 67

H (editor command) 61
headerless code 123
HELL-FREEZES-OVER example 43
HERE (FORTH word) 45, 51, 106
HEX (FORTH word) 50, 106
HLD (FORTH word) 23, 106
HOLD (FORTH word) 53, 107

I (editor command) 61
(FORTH word) 38, 107
ID. (FORTH word) 107
IF (FORTH word) 35, 107
IMMEDIATE (FORTH word) 30, 107
example 30, 32
words compilation of 31
IN (FORTH word) 23, 107
indefinite loops 42
index check for arrays 71
index to manual 113
indirect threaded code 4
input 45
character 45
INPUT example 47
input numeric 46
text 45
INSCR example 80
integers printing 11
intelligent CMOVE 49
INTERPRET (FORTH word) 107
introduction to FORTH 2
to manual 1
INVERT (graphics word) 66
INVERT editing example 62
IP location 25, 26

J (FORTH word) 41
JTEST example 41

K (FORTH word) 42
KEY (FORTH word) 107
KEYBOARD example 83

LATEST (FORTH word) 77, 107
LEAVE (FORTH word) 40, 108
LEFT$ example 51
LFA (FORTH word) 20, 108
LIMIT (FORTH word) 108
LINE (graphics word) 67
line-drawing graphics 66
lines deleting 61
editing 61
erasing 61
replacing 61
link field address 19
LIST (FORTH word) 56
example 73
LIT (FORTH word) 108
LITERAL (FORTH word) 32, 108
LOAD (FORTH word) 56, 108
loading editor 55
FORTH 7
screens 56
tape interface 55
locations:
IP 25, 26
N 25
UP 25
W 25, 26
XSAVE 25, 25
zero-page 25
logical operators 15
LOOK-UP example 40, 41
LOOP (FORTH word) 37, 108

loop index 37, 38
  limit 37
loops indefinite 42
  nested 39
loudspeaker 82

M (editor command) 63
M* (FORTH word) 16, 108
M/ (FORTH word) 17, 109
M/MOD (FORTH word) 17, 109
machine-code 24
manipulating blocks of memory 48
MATCH (editor command) 63
MAX (FORTH word) 15, 109
MD* example 17
MEMBER example 69
memory allocation 125
  extension 67, 127
  graphics 65
memory map ATOM FORTH 125, 126
  System FORTH 119
memory usage 65
MESSAGE (FORTH word) 109
meta-FORTH 69
MIN (FORTH word) 15, 109
MINUS (FORTH word) 12, 109
mixed-precision operators 16, 17
MOD (FORTH word) 12, 109
mode 4 graphics 67
modes graphics 65
monitor commands 81
MONITOR example 81
MOVE (graphics word) 67
music 83

N (editor command) 63
  location 25
name field address 19
names of FORTH words 89, 122
nested branches 37
  loops 39
NFA (FORTH word) 20, 109
NOOP (FORTH word) 109
notation postfix 11
  reverse-Polish 11
NUCTOP (FORTH word) 109
NUMBER (FORTH word) 46, 109
numbers compilation of 32
  double-precision 16
  single-precision 11
numeric bases 49
  conversion 47
  input 46
  output 51
  output formatting 53
NUMIN example 46

OCTAL 50
one-dimensional arrays 71
operators:
  double-precision 16
  logical 15
  mixed-precision 16, 17
  relational 15
  single-precision 12
  stack 13, 17
OR (FORTH word) 15, 109
OS commands 81
OSDATA (FORTH word) 110
OSLOAD (FORTH word) 110
OUT (FORTH word) 23, 110
output 50
  character 50
  formatting numeric 53
  numeric 51
  text 50
OUTSCR example 80
OVER (FORTH word) 13, 110

P (editor command) 60, 61
PAD (FORTH word) 54, 61, 110
parameter field 19
  stack 10
PAUSE example 42
PFA (FORTH word) 20, 110
PICK (FORTH word) 13
PLOT (graphics word) 66
PLOT-IT example 86
point-plotting graphics 66
POPTWO 24
postfix notation 11
printing integers 11
PROGRAM (FORTH word) 58
pronunciation of FORTH words 89

quadratic example 12, 14
QUERY (FORTH word) 45, 110
QUICKSORT example 78
QUIT (FORTH word) 110

R (editor command) 61
  (FORTH word) 13, 110
R£ (FORTH word) 23, 111
R0 (FORTH word) 23, 110
R> (FORTH word) 13, 111
reading further 129
RECTANGLE example 72
recursion 77
  example 77
references 129
REL (graphics word) 67
relational operators 15
relative plotting graphics 67
relocation dictionary 127
  tape buffer 127
  user variable area 127
REPEAT (FORTH word) 42, 43, 111
replacing lines 61
resolution graphics 65
return stack 10, 13

reverse-Polish notation 11
RIGHT$ example 51
RMOVE (graphics word) 67
RND example 59
ROLL (FORTH word) 13, 111
ROT (FORTH word) 13, 111
RP! (FORTH word) 111
RP@ (FORTH word) 111
RPLOT (graphics word) 67

S (editor command) 61
S->D (FORTH word) 111
S0 (FORTH word) 23, 111
SAVE (tape interface word) 57
saving dictionary space 122
SCR (FORTH word) 23, 57, 111
screen clear 67
  copying utility 79
  numbering 63
screens 55, 58
  loading 56
security compiler 31
separating applications 21
SEQUENCE example 39
SETXY (graphics word) 67
SHOWASCII example 45
SIGN (FORTH word) 53, 112
single-precision arithmetic 11
  numbers 11
  operators 12
SIZE example 70
SMUDGE (FORTH word) 16, 24, 112
sort example 78
sound 82
SP! (FORTH word) 112
SP@ (FORTH word) 112
SPACE (FORTH word) 112
SPACES (FORTH word) 112
stack actions of FORTH words 89
stack computation 10
  operators 13, 17
  overflow 38
  parameter 10
  print 88
  return 10, 13
  transfers 13
stacks 9
STARS example 7
STATE (FORTH word) 23, 112
status of FORTH words 89
storage of strings 50, 51
STRING example 73
string handling 51
strings 73
  editing 62
STRINGS example 51
strings storage of 50, 51
SWAP (FORTH word) 13, 112
System FORTH 119

T (editor command) 61
tables 72
tape buffer relocation 127
tape interface 56
  loading 55
TENCOUNT example 37
terminating jumps 25
text FORTH words in 1
  input 45
  input delimiter 46
  output 50
THEN (FORTH word) 35, 112
threaded code 4
TIB (FORTH word) 23, 112
TILL (editor command) 62
timer 84
TOGGLE (FORTH word) 15, 113
tones example 82
TRAVERSE (FORTH word) 113
TRIANGLE example 67
two's-complement arithmetic 117
two-dimensional arrays 72
TYPE (FORTH word) 45, 50, 51, 113

U* (FORTH word) 16, 113
U. (FORTH word) 11, 52, 113
U/ (FORTH word) 17, 113
U< (FORTH word) 15, 113
UNTIL (FORTH word) 42, 113
UP location 25
USER (FORTH word) 22, 113
user variable area relocation 127
user variables 23
utilities:
  MONITOR 81
  screen copying 79

validity of FORTH words 4
VALUES example 71
VARIABLE (FORTH word) 22, 114
  example 70
VIA timer 84
VLIST (FORTH word) 7, 114
VOC-LINK (FORTH word) 23, 114
vocabularies:
  EDITOR 58
  FORTH 29
  GRAPHICS 66
VOCABULARY (FORTH word) 19, 28, 114

W location 25, 26
WAIT example 81
WARM (FORTH word) 8
warm start 7
warning 91
WARNING (FORTH word) 23, 46, 114
WASHING example 4
WHERE (editor command) 57
WHILE (FORTH word) 42, 43, 114

WHITE (graphics word) 66
WIDTH (FORTH word) 23, 115, 122
WORDS example 73

X (FORTH word) 115
  (editor command) 62
X-register 25
XDIR (graphics word) 67
XOR (FORTH word) 15, 115
XSAVE location 25, 25

YDIR (graphics word) 67

zero-page location 25

[(FORTH word) 30, 115
[COMPILE] (FORTH word) 30, 115
  example 31, 32

] (FORTH word) 30, 115

ACORNSOFT

SECOND EDITION

ISBN 0 907876 05 6