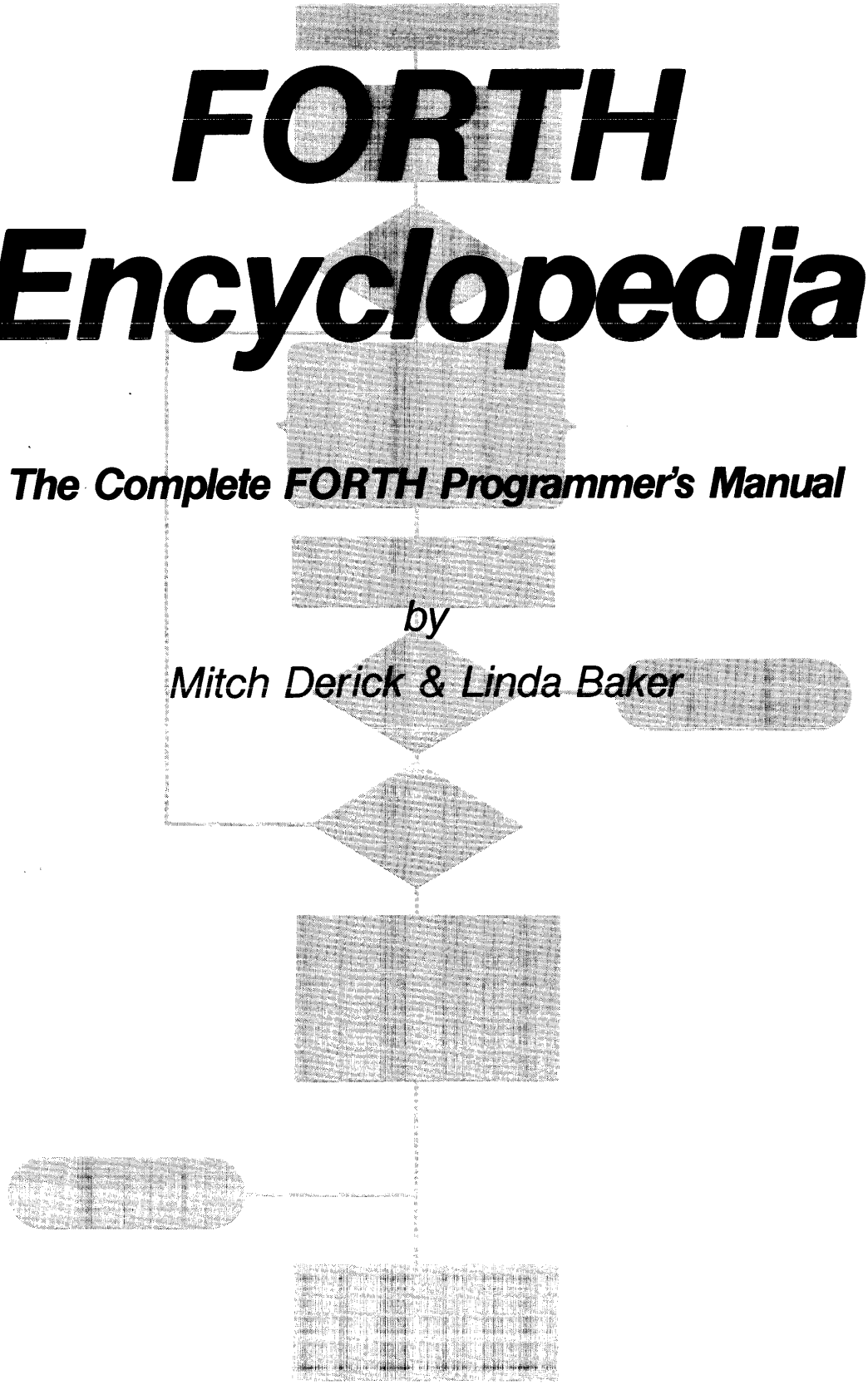
FORTH Encyclopedia

The Complete FORTH Programmer's Manual

by

Mitch Derick & Linda Baker



Second Edition

Mountain View Press, Inc.

FORTH Encyclopedia

The Complete FORTH Programmer's Manual

by

Mitch Derick & Linda Baker

Second Edition

Mountain View Press, Inc.

Copyright © 1982

by

Mitch Derick and Linda Baker

Mountain View Press
PO Box 4656
Mountain View, CA 94040

TABLE OF CONTENTS

! 1
!CSP 2
..... 3
#> 5
#S 6
' 7
(..... 9
(.") 10
(+LOOP) 12
(;CODE) 13
(ABORT) 14
(DO) 15
(FIND) 16
(LINE) 19
(LOOP) 21
(NUMBER) 22
* 25
*/ 26
*/MOD 27
+ 28
+! 29
+- 30
+BUF 31
+LOOP 33
+ORIGIN 35
, ... 37
- 38
--> 39
-DUP 41
-FIND 42
-TRAILING 44
. 46
." 47
.LINE 49
.R 50
/ 51
/MOD 52
0 53
0< 54
0= 55
0BRANCH 56
1 57
1+ 58
2 59
2+ 60
3 61
: 62
; 65
;CODE 67
;S 70
< 71
<# 72
<BUILDS 73
= 76
> 77
>R 78
? 79
?COMP 80
?CSP 81
?ERROR 82
?EXEC 83
?LOADING 84
?PAIRS 85
?STACK 86
?TERMINAL 88
@ 89
ABORT 90
ABS 91
AGAIN 92
ALLOT 94
AND 95
B/BUF 96
B/SCR 97
BACK 98

BASE 99
BEGIN 100
BL 102
BLANKS 103
BLK 104
BLOCK 105
BRANCH 109
BUFFER 110
C! 114
C, ... 115
C/L 116
C@ 117
CFA 118
CMOVE 119
COLD 120
COMPILE 122
CONSTANT 124
CONTEXT 126
COUNT 127
CR 128
CREATE 129
CSP 132
CURRENT 133
D+ 134
D+- 135
D. 136
D.R 137
DABS 139
DECIMAL 140
DEFINITIONS 141
DIGIT 142
DLITERAL 144
DMINUS 146
DO 147
DOES> 149
DP 153
DPL 154
DPUSH 155
DR0 156
DR1 156
DROP 157
DUP 158
ELSE 159
EMIT 162
EMPTY-BUFFERS 163
ENCLOSE 164
END 168
ENDIF 170
ERASE 172
ERROR 173
EXECUTE 175
EXPECT 176
FENCE 180
FILL 181
FIRST 182
FLD 183
FLUSH 184
FORGET 186
FORTH 188
HERE 189
HEX 190
HLD 191
HOLD 192
HPUSH 193
I 194
ID. 195
IF 197
IMMEDIATE 199
IN 200
INDEX 201
INTERPRET 203
IP 208
KEY 209
LATEST 210
LEAVE 211
LFA 212
LIMIT 213
LIST 214

LIT 216
LITERAL 217
LOAD 219
LOOP 221
M* 223
M/ 224
M/MOD 226
MAX 228
MESSAGE 229
MIN 231
MINUS 232
MOD 233
MON 234
NEXT 235
NFA 236
NULL 237
NUMBER 240
OFFSET 244
OR 245
OUT 246
OVER 247
PAD 248
PFA 249
PREV 250
QUERY 251
QUIT 252
R 254
R# 255
R/W 256
R0 259
R> 260
REPEAT 261
ROT 264
RP! 265
RP@ 266
S->D 267
S0 268
SCR 269
SIGN 270
SMUDGE 271
SP! 272
SP@ 273
SPACE 274
SPACES 275
STATE 276
SWAP 277
TASK 278
THEN 279
TIB 281
TOGGLE 282
TRAVERSE 283
TRIAD 285
TYPE 287
U* 289
U. 291
U/ 292
UNTIL 295
UPDATE 297
USE 298
USER 299
VARIABLE 301
VLIST 303
VOC-LINK 305
VOCABULARY 306
W 314
WARNING 315
WHILE 316
WIDTH 318
WORD 319
X 322
XOR 323
[..... 324
[COMPILE] 325
] 327

FORTH SYSTEM MESSAGES 328
ASCII CHARACTER SET (7-bit code) 329
STANDARD fig-FORTH MEMORY MAP 330
ALPHABETICAL INDEX 331
FUNCTIONAL INDEX 332

PREFACE

The purpose of the FORTH ENCYCLOPEDIA is to make available in one location all of the information necessary to use and understand each individual word in the FORTH language. It is useful for everyone from FORTH beginners to FORTH experts. Just like any other programmer's manual, this book is a tool which allows the programmer to spend more time "solving the problem" and less time "fighting the language."

As FORTH programmers, we saw a need for documentation that would explain in one place everything one needed to know about a FORTH word. It is our intent to relieve the programmer of the burden of sifting through multiple sources and/or FORTH code in order to understand a desired FORTH definition.

The specific implementation documented is the 8080 Version 1.1 (CP/M) fig-Model. However, the usefulness of this book is not limited to just this implementation. Indeed, even non-fig-FORTH programmers will find it a useful reference. The activity of each definition is both literally and conceptually described. Each definition references FORTH-79. Those words which have two or more activities (defining words and compiler words) have all activities described in detail. The low-level code primitives are described in "generic" terms common to all assembly languages.

In summary, this book is truly a FORTH "Encyclopedia" as it provides information ranging from the overview to the internal details of each definition. The acceptance of FORTH in the software community has been limited by a lack of this type of documentation. We hope this and other recent publications will help satisfy that need.

ACKNOWLEDGMENTS

The efforts of many people (both directly and indirectly) made this book possible. First and foremost we would like to thank Eric Weaver who worked closely with us and edited both the rough and final copies of the manuscript. Without Eric's expert knowledge of the internals of FORTH, the FORTH ENCYCLOPEDIA would not have attained its high degree of technical quality.

We would also like to thank Gary Fierbach, Jim McDaniels, Terry Holmes, and Kim Harris for reviewing the final manuscript.

We greatly appreciate the "above and beyond" efforts of Dick Weismann and Ken Moore in the physical preparation of the manuscript for printing.

We would especially like to thank Kim Harris for teaching us how to use FORTH and for allowing us to reproduce some of his class materials in this publication.

Then there is Fig. Thank you, FORTH Interest Group. Although not directly involved in this book, the indirect efforts of many members of the FORTH community were what made all of this possible in the first place. After analyzing every single piece of code in the fig-FORTH Model, we truly appreciate and acknowledge the magnitude of effort Bill Ragsdale expended in writing the first fig-Model.

We would like to thank the FORTH Implementation Team, especially John Cassady (who implemented the 8080 Version) and Kim Harris (who later modified the 8080 Version 1.1 referenced in this book).

Last but certainly not least, thank you Charles Moore for creating such a marvelous language.

The fig-FORTH Model and the 8080 Version 1.1 listing are distributed by:

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

Each word (or "definition") description follows this general format:

1. Word Name (Parameter Stack Activity)
2. Text Section
   a. Word Pronounciation
   b. General Description
   c. Detailed Description
   d. Example
3. Formal Parameters
4. High or Low Level Statement
5. Likely Error Messages
6. "Refer to" List
7. FORTH-79
8. Flowchart Section
   a. High Level Flowchart (if applicable)
   b. Detailed Flowchart

1. **Word Name (Parameter Stack Activity)** -- The first line of each description is printed in bold characters. The definition name is on the left followed by the parameter stack activity enclosed in parentheses. The general format for parameter stack entries is:

   (before execution -- after execution)

   When multiple stack parameters are listed, each entry is separated by a "\". The lefthand entry is lowest on the stack, the righthand entry is the top of the parameter stack.

   HINT: By pronouncing "\" as "under" and "--" as "leaves" this horizontal stack format is quite readable. For example:

   (value1 \ value2 -- value3) becomes "value1 under value2 leaves value3"

Some definitions return different stack conditions depending upon their input (for example, successful or unsuccessful completion of a word). In these cases, each condition is listed separately.

Normally the stack parameters described reflect the "execution time" (Sequence 3) action of a word. (See "Definition of Sequence Times" Section.) If a word has more than one action (e.g., execution time and compile time action), the stack parameters for each activity are listed separately.

2. **Text Section**

   a. **Word Pronounciation** - Since FORTH is intended to be a speakable language, the pronunciation (if necessary) is given inside parentheses.

   b. **General Description** - The first part of the text section is usually a summary of the purpose of the definition.

   c. **Detailed Description** - This section is the pertinent information concerning the word.

      Compiler and defining words have two sets of actions and therefore the description of this category of words is divided into two sections. The compile time action is listed followed by the execution time action.

   d. **Example** - An example of a word which uses the described word is usually included.

3. **Formal Parameters** - This section contains a description of the "at entry" and "at exit" parameter stack entries for the definition. This normally reflects the execution time action of a word. Words with more than one action have parameters for each action listed separately.

   If no stack parameters are present, "at entry" or "at exit" is followed by the phrase "No parameters".

   The activity of some words affect more than just the parameter stack. In those instances, all other affected parameters are also described.

4. **High or Low Level Statement** - This states whether the word is a high level definition (comprised of other FORTH definitions) or an assembly language code primitive.

5. **Likely Error Messages** - This section describes the error messages which are most likely to occur when using this word. (Only the most likely messages are listed because serious error conditions can cause indeterminable error messages.)

6. **"Refer to" List** - Lists words to refer to for more information.

7. **FORTH-79** - This shows how the described word relates to the FORTH-79 Standard.

8. **Flowchart Section** - All flowcharts use the following format:

The flowcharts are generally divided into three columns. The leftmost column contains word names and labels. The middle column contains a formal definition of the activity of the word. The rightmost column contains branch labels and comments describing the activity of this word as it relates to the definition being discussed.

A right square bracket, located on the far right of the comment column, encloses those words which would normally be grouped together as a FORTH phrase. A phrase in FORTH is a logical grouping of words which when combined together produce a desired result.

The larger, more complex words may also be divided into groupings analogous to sentences in English. These sentences are proceeded by a comment enclosed in curly brackets.

The more complex flowcharts are accompanied by a high level (macro) flowchart so the overall action of the definition can easily be grasped. Each box in the macro flowchart roughly corresponds to a curly bracket section of the detailed flowcharts.

The same general format is followed for the low level code primitives. Code definitions must always terminate with an eventual jump to NEXT . This is shown with a dashed line extending across all three columns.

ABORT vs. QUIT:

Either an ABORT or a QUIT may occur when an error condition arises. The contents of the user variable WARNING determines which action is taken. Flowchart references assume that a QUIT will occur.

DEFINITION OF SEQUENCE TIMES

One of the major philosophical differences between FORTH and the more traditional programming languages is the fact that the FORTH compiler is itself composed of FORTH words. This compiler may also be extended at any time with the addition of new compiler words; this causes a special set of problems to arise.

Specifically:

What is a word doing at any given time?

Is it being defined?

Is it being compiled?

Is it executing?

This ambiguity is made clear through the use of the terms Sequence 1, Sequence 2, and Sequence 3. These sequences will be described in ascending order of complexity.

Sequence 3:

The purpose of all FORTH definitions is to eventually execute. The act of executing is termed a definition's "run time" or "execution time" or "Sequence 3" action. For example, dropping a value from the top of the parameter stack is the "run time" (Sequence 3) action of the word DROP .

Sequence 2:

In order to be able to execute, a definition must first be compiled or assembled. The act of being compiled or assembled, then, is the "Sequence 2" time for the word being compiled. e.g., The Sequence 2 time for DROP is when DROP is being compiled.

Confusion often arises when one considers the action of the compiler words themselves. Compiler words must execute in order to compile a definition. i.e., Compiler words execute (the compiler word's Sequence 3 time) during the Sequence 2 time of the word being compiled.

Compiler words normally have two sets of actions described; those at compile time (Sequence 2) and those at execution time (Sequence 3). The most common compiler words are control structure words such as ELSE . ELSE is used within a definition (FROG for example) to produce a specific action when that word, FROG, is executed at its Sequence 3 time. This action is the Sequence 3 action of ELSE . But, in order for ELSE to be able to correctly perform this desired action, certain manipulations must be performed when FROG is being compiled (at FROG's Sequence 2). The code within ELSE which performs these compile time manipulations determines the Sequence 2 action of ELSE . This dual action is true for all compiler words.

Sequence 1:

There are groups or types of words in FORTH which are defining words. (Defining words are generally classified as words which do not execute in the system "compile" state and which directly use the word CREATE (and not :) to begin creating a definition.) These words create compiler definitions, or "parents", which when executed at compile time (Sequence 2) create "children" definitions which then execute (Sequence 3) to perform some task. "Parent" definitions (e.g., VOCABULARY) are created at the parent's Sequence 1 time. "Child" definitions (e.g., FORTH) are compiled via the "parent" at the "child's Sequence 2 time. The "child" then executes at its own Sequence 3 time (e.g., when FORTH executes, FORTH is made the CONTEXT vocabulary).

Always bear in mind that the Sequence time of any definition is relative to itself. It is perfectly legal and necessary for compiler words to be executing (Sequence 3) while a definition is being created (Sequence 1 or Sequence 2).

Some compiler words purposely switch the system back and forth between states while all the time remaining in the definition-being-compiled's Sequence 2 time. At any given time the "state" the system is in may be either "interpretation" state or "compilation" state. Do not confuse the overall Sequence time action of a definition with the "state" of the system.

! (value \ address --)

! (pronounced "store") stores the 16-bit value located in the second parameter stack entry into the memory location specified by the top parameter stack entry.

C! is the word used to store 8-bit (or byte or character) values.

@ has the opposite effect of ! .

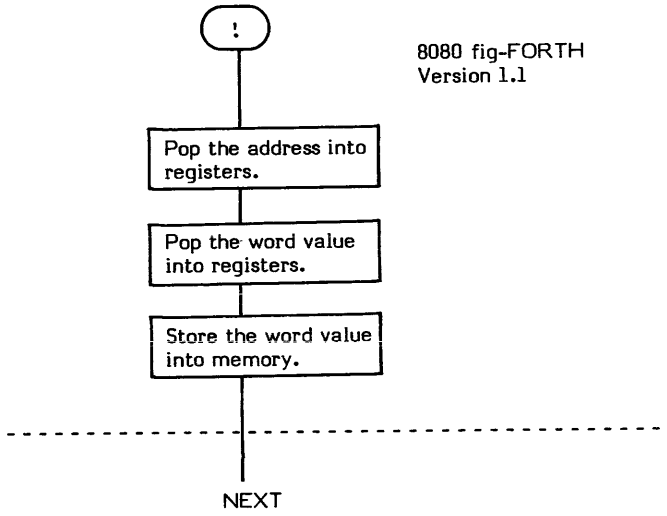
* **At entry** - The top of the parameter stack contains the 16-bit address specifying the memory location the value is to be stored into. The second stack entry contains the 16-bit value.

* **At exit** - No parameters.

! is a low level code primitive.

Refer to C! , and @ .

FORTH-79: The FORTH-79 equivalent for ! is ! .



!CSP

!CSP (–)

!CSP (pronounced "store-C-S-P") saves the current parameter stack address in the user variable CSP . This word is used in conjunction with ?CSP to determine if the parameter stack has become unbalanced due to a compilation error.

An example of the use of !CSP can be found in : .

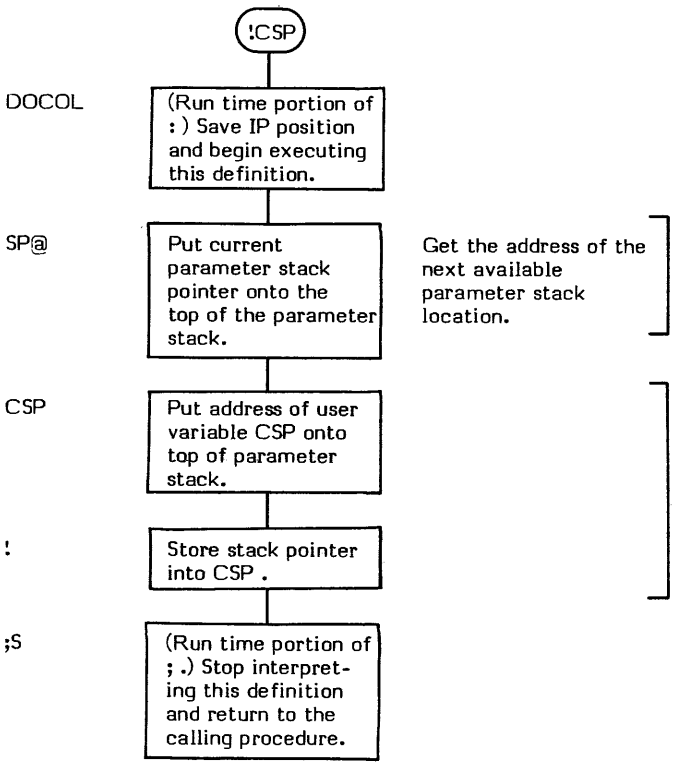
* **At entry** - No parameters.
* **At exit** - No parameters.

!CSP is a high level colon definition.

Refer to ?CSP .

FORTH-79: There is no FORTH-79 equivalent for !CSP .

Definition: : !CSP (--) SP@ CSP ! ;



(double precision value — double precision quotient)

(pronounced "sharp") performs a binary-to-ascii conversion of one digit of a double precision value into an ascii character that is placed into a pictured numeric output string. This output string is built downward starting from one byte before PAD . The conversion proceeds from right to left (low order digit to high order digit) with one "column" being converted each time # is executed. That is, the first time # is executed, the units digit is converted; the next, the tens digit; the next, the hundreds digit; etc.

One digit is converted each time # is executed. # uses the current BASE value as the conversion radix. The double precision value is divided by the radix. The remainder is converted to an ascii digit; the quotient remains on the stack. <# must first be executed to set up for # . # is used in the general form <# # #> to convert single digits. The form <# # # ... #> is used to convert as many digits as specified. (Refer to #S for converting an entire double precision number at one time.)

A double precision input value of zero simply results in an ascii output digit of zero.

D.R is an example of a word that uses pictured numeric output. #S is an example of a word that uses # .

* **At entry** - The top of the parameter stack contains an unsigned 32-bit double precision value with the high order portion in the first stack entry and the low order portion in the second entry.

  Note: # will actually only convert a positive double precision value (i.e., only 31 bits). Therefore a DABS should normally precede the <# in a pictured numeric expression to convert negative double precision values to their absolute value. SIGN is used to display the sign.

  BASE contains the radix to be used in conversion.

* **At exit** - The top of the parameter stack contains the quotient of the original value divided by BASE in the form of a 32-bit unsigned double precision value with the high order portion in the first stack entry and the low order portion in the second stack entry.

Characters will be placed into memory from high to low memory starting at the beginning of PAD and working toward the end of the dictionary. HLD points to the last character converted.

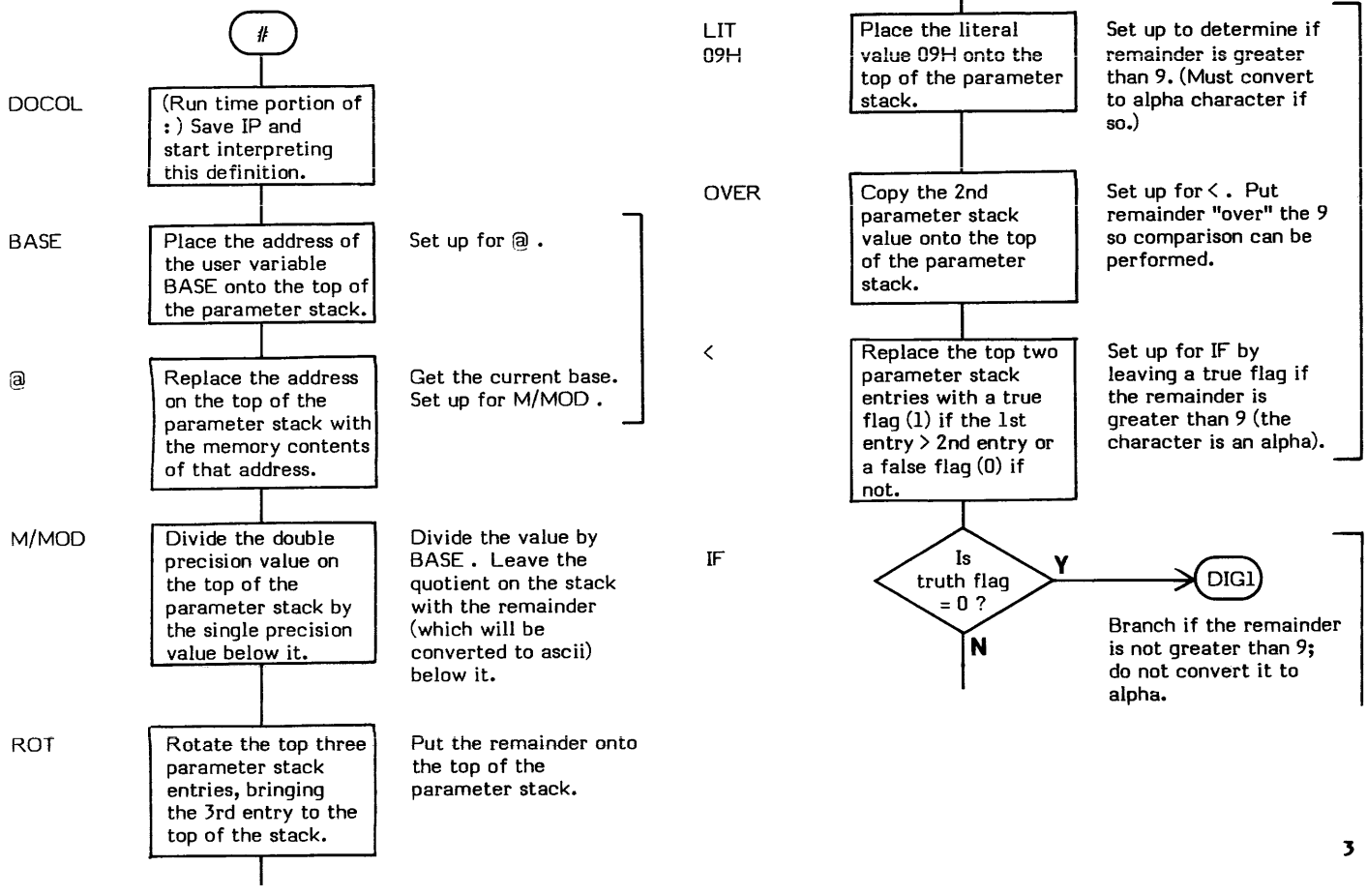
is a high level colon definition.

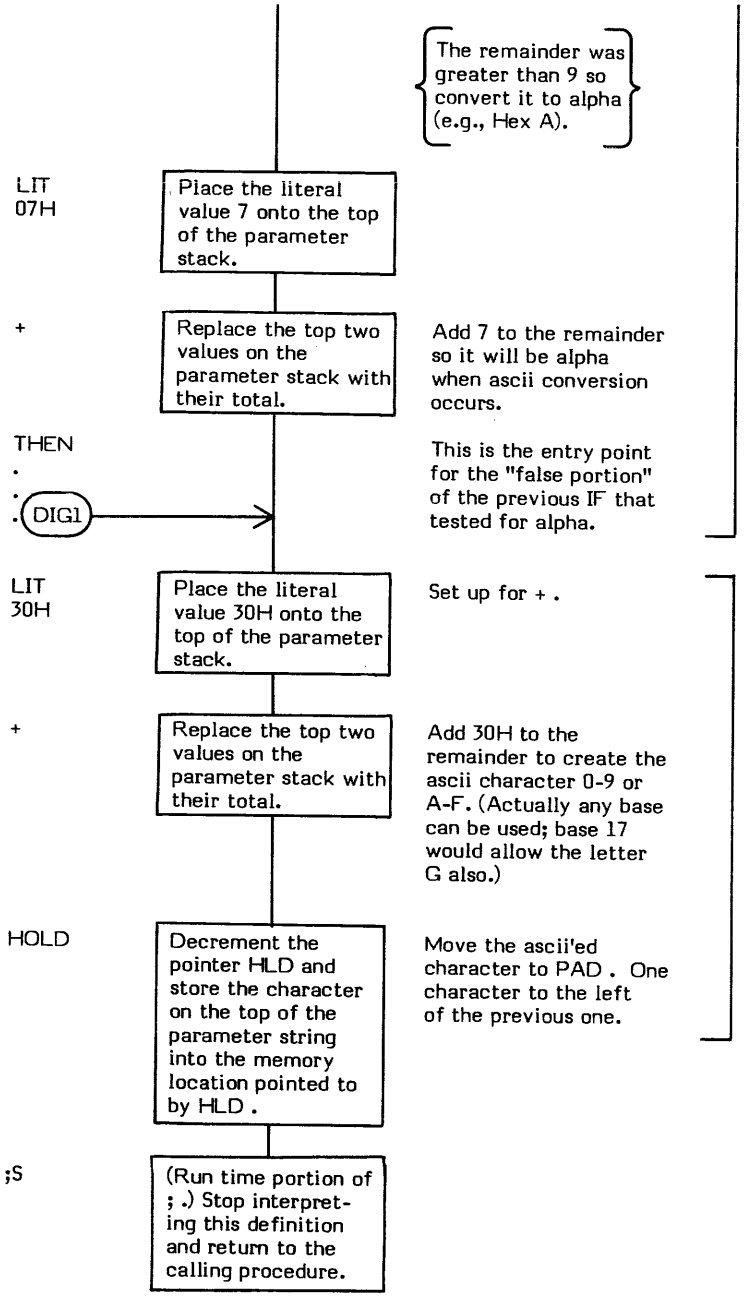
Refer to <# , #> , #S , HOLD , SIGN , HLD , BASE , and PAD .

FORTH-79: The FORTH-79 equivalent for # is # .

Definition:

```
:  #  ( Dvalue - Dquotient )
     BASE @   M/MOD ROT   9 OVER <
     IF 7 +   THEN   30 + HOLD   ;
```




The remainder was greater than 9 so convert it to alpha (e.g., Hex A).
LIT
07H
Place the literal value 7 onto the top of the parameter stack.
+
Replace the top two values on the parameter stack with their total.
Add 7 to the remainder so it will be alpha when ascii conversion occurs.
THEN
.
.
.
DIG1
This is the entry point for the "false portion" of the previous IF that tested for alpha.
LIT
30H
Place the literal value 30H onto the top of the parameter stack.
Set up for + .
+
Replace the top two values on the parameter stack with their total.
Add 30H to the remainder to create the ascii character 0-9 or A-F. (Actually any base can be used; base 17 would allow the letter G also.)
HOLD
Decrement the pointer HLD and store the character on the top of the parameter string into the memory location pointed to by HLD .
Move the ascii'ed character to PAD . One character to the left of the previous one.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

#> **(double precision value — address\count)** **#>**

#> (pronounced "sharp-greater-than") terminates a pictured numeric conversion expression. #> drops the double precision conversion value left by # and replaces it with the beginning address of the converted character string and the string length (in a format suitable for TYPE).

#> is used in the general form:

<# # #>

D.R is an example of a word that uses pictured numeric output.

* **At entry** - The top of the parameter stack contains a double precision value occupying the first and second stack entries. HLD points to the beginning of a converted pictured numeric character string.

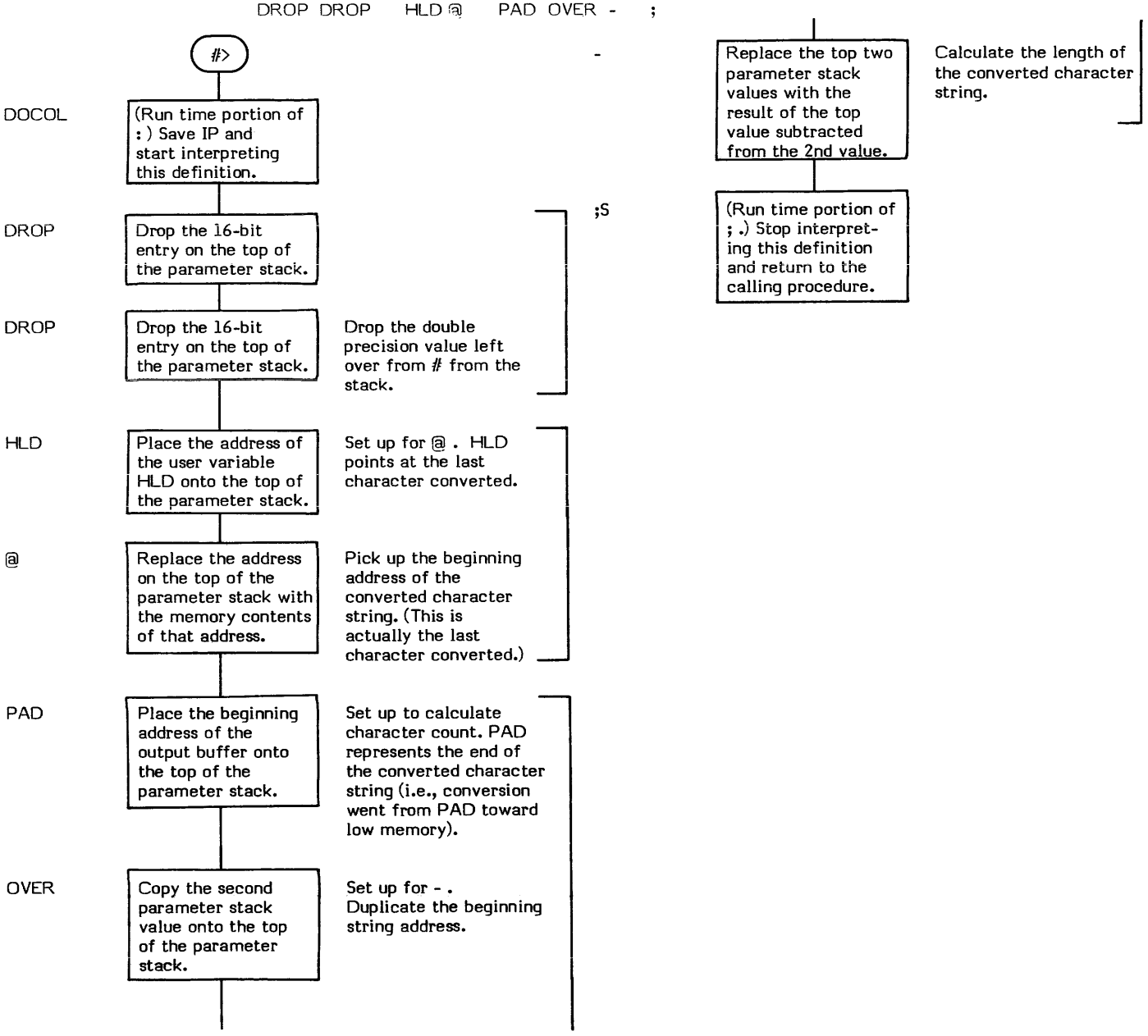
* **At exit** - The top of the parameter stack contains an address pointing to the beginning of the converted pictured numeric character string. The second parameter stack entry contains the character count of the string.

#> is a high level colon definition.

Refer to <# , # , #S , HOLD , HLD , SIGN , and PAD .

FORTH-79: The FORTH-79 equivalent for #> is #> .

Definition: : #> (Dvalue - addr\count)
DROP DROP HLD @ PAD OVER - ;



#S

#S (unsigned double precision value — 0\ 0)

#S (pronounced "sharp-S") performs a binary-to-ascii conversion of a double precision value into an ascii pictured numeric output character string. This character string is created in memory, starting at one byte before PAD and working downward towards low memory. The word performs a # conversion until the original double precision value is completely converted to ascii. Actually # is called repeatedly until conversion is complete (i.e., the resulting quotient is double precision zero).

#S is used within a <# #> expression. D.R is an example of a word that uses #S for pictured numeric output.

* **At entry** - The top of the parameter stack contains an unsigned 32-bit double precision value which is to be converted from binary to ascii. The high order portion is in the first stack entry and the low order portion is in the second entry.

  BASE contains the radix to be used in the conversion.

* **At exit** - The first and second parameter stack entries contain a double precision value of zero.

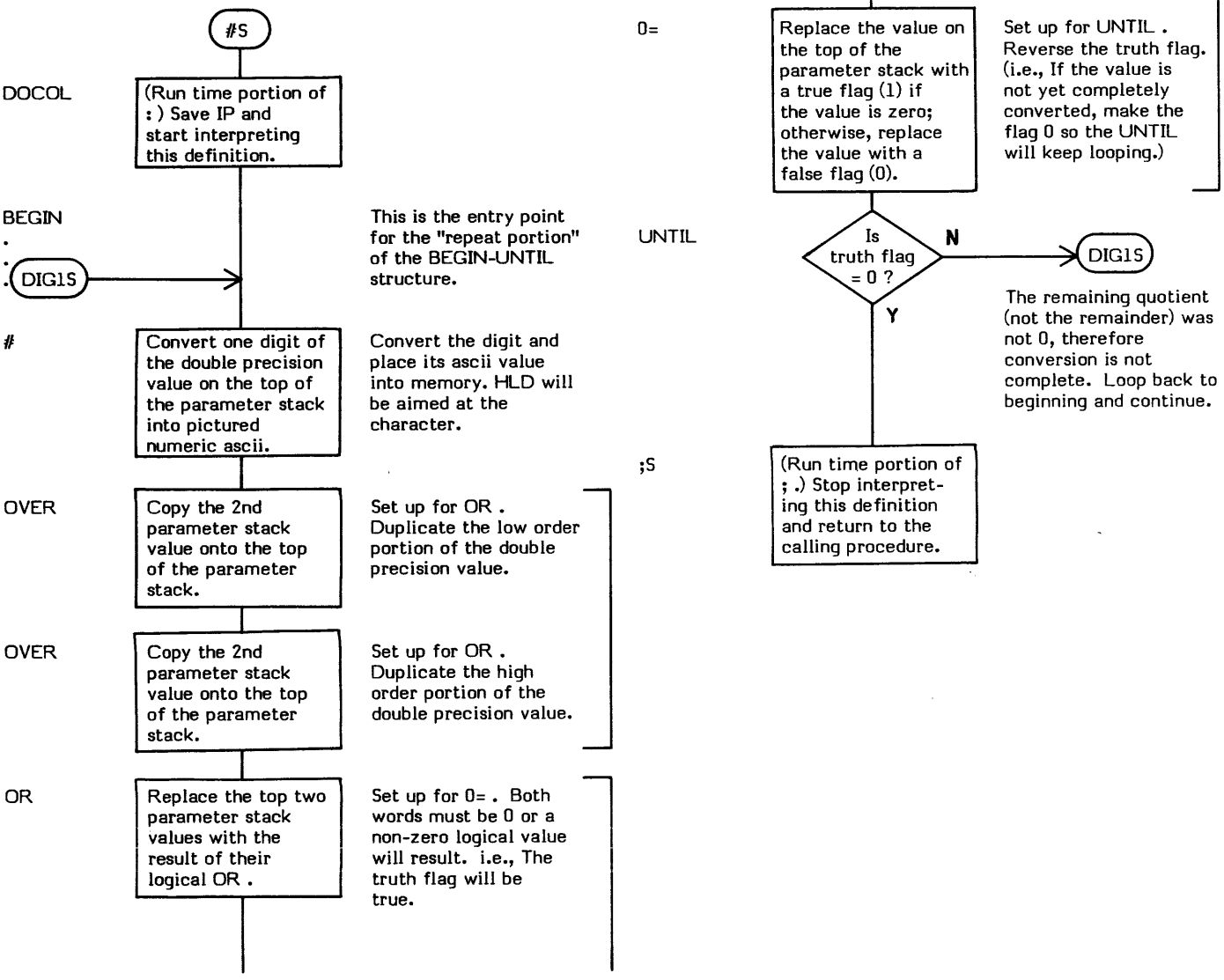
Converted ascii characters are located in memory. HLD points to the last character converted.

#S is a high level colon definition.

Refer to <# , # , #> , SIGN , HOLD , HLD , and PAD .

FORTH-79: The FORTH-79 equivalent for #S is #S .

Definition: : #S (Dunsigned value - 0\ 0)
BEGIN # OVER OVER OR 0= UNTIL ;



COMPILE STATE: (—)
(Sequence 2)

EXECUTION STATE: (— address of specified word)
(Sequence 3)

' (pronounced "tick") in execution state, ' places the Parameter Field Address of a specified definition onto the top of the parameter stack. In compilation state it compiles the address into the dictionary. It is used in the form:

' nnnn

where nnnn is the name of the desired word.

Since LITERAL is used within the definition, the word exhibits two characteristics:

1. If the system is in compilation state, the address is compiled into the dictionary as a literal.

2. If the system is in execution state, the address is left on the top of the parameter stack.

' searches both the CONTEXT and CURRENT vocabularies. If the specified word is not found, Error Message 0 ("?") is issued and a QUIT occurs.

-FIND is the basis of ' since it performs the necessary dictionary search.

Note that ' is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE STATE (Sequence 2):

* **At entry** - No parameters.

* **At exit** - The address is compiled as a literal into the definition being compiled. No parameters on the paramater stack.

EXECUTION STATE (Sequence 3):

* **At entry** - No parameters.

* **At exit** - The address of the specified word is on the top of the parameter stack.

LIKELY ERROR MESSAGES:

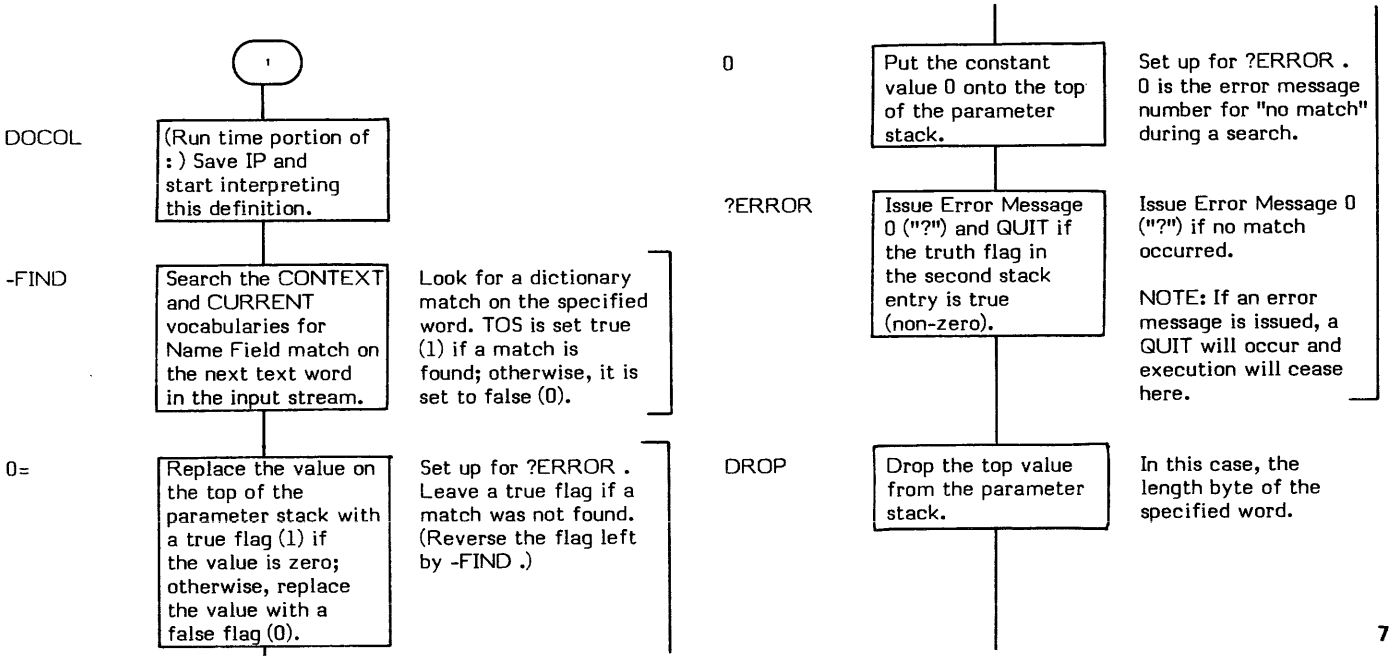
? pronounced "HUH?" (0) — The word in question cannot be found in the dictionary.

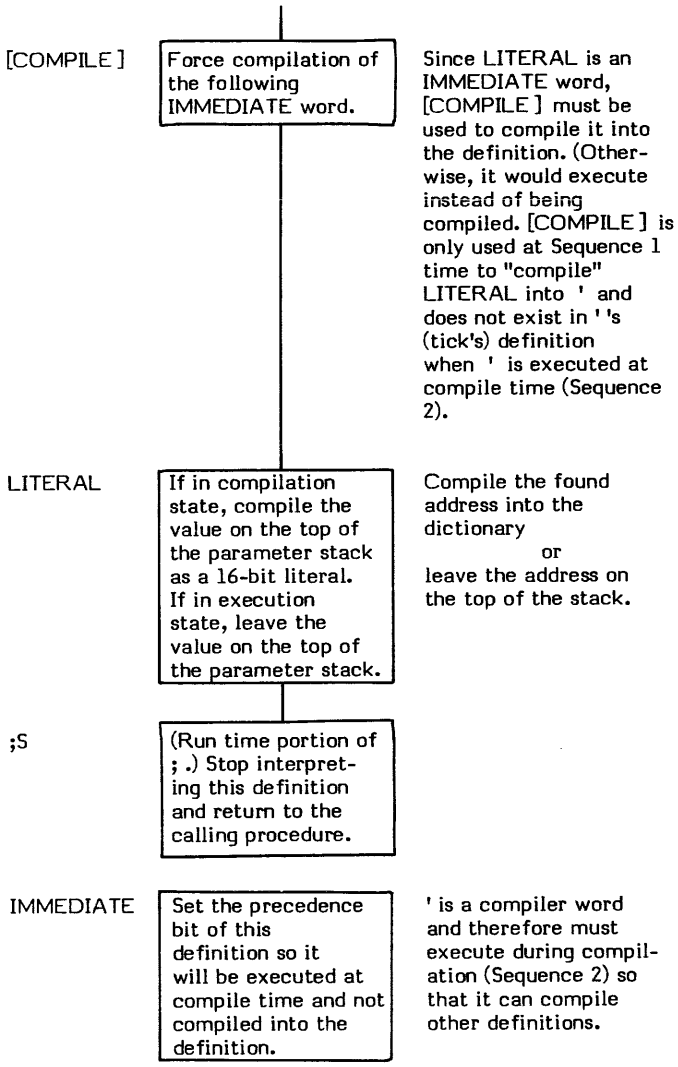
' is a high level colon definition.

Refer to -FIND .

FORTH-79: The FORTH-79 equivalent for ' is '.

Definition: : ' (— addr)
-FIND 0= 0 ERROR DROP [COMPILE] LITERAL ; IMMEDIATE




[COMPILE]
Force compilation of the following IMMEDIATE word.
Since LITERAL is an IMMEDIATE word, [COMPILE] must be used to compile it into the definition. (Otherwise, it would execute instead of being compiled. [COMPILE] is only used at Sequence 1 time to "compile" LITERAL into ' and does not exist in ' 's (tick's) definition when ' is executed at compile time (Sequence 2).
LITERAL
If in compilation state, compile the value on the top of the parameter stack as a 16-bit literal. If in execution state, leave the value on the top of the parameter stack.
Compile the found address into the dictionary
or
leave the address on the top of the stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of this definition so it will be executed at compile time and not compiled into the definition.
' is a compiler word and therefore must execute during compilation (Sequence 2) so that it can compile other definitions.

((–)

((pronounced "left paren") is used to denote the beginning of a comment. It is used in the form:

(comment)

Note that (is a FORTH word and therefore must be separated from the comment string by at least one blank. The comment must be terminated by a) ("right paren"). The terminating right parenthesis does not have to be preceded by a blank. Comments may appear inside or outside of a definition.

The basis of (is WORD . WORD is given the ascii value of ")" for an input delimiter parameter. WORD then keeps reading (and ignoring) text until encountering a ")" character.

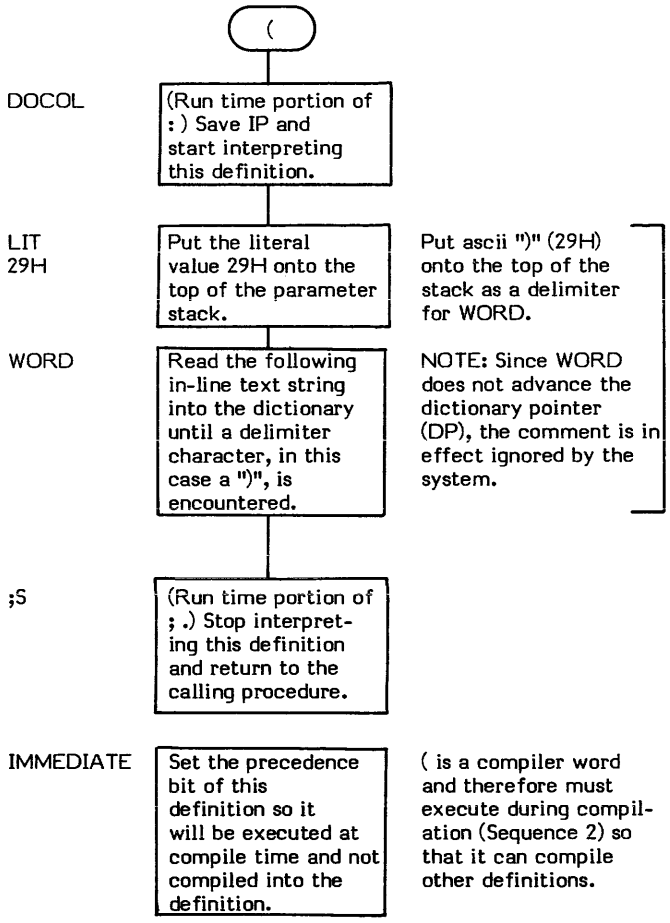
Note that (is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

* **At entry** - No parameters.
* **At exit** - No parameters.

(is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for (is (.

Definition: : ((–)
29 WORD ; IMMEDIATE



(.")

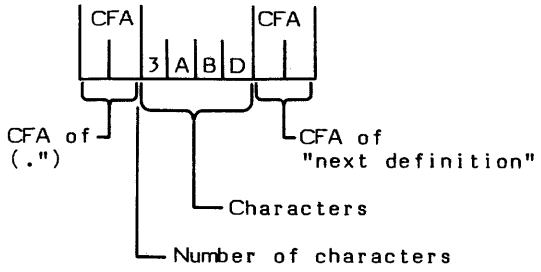
(.") Return Stack (address — address)

(.") (pronounced "paren-dot-quote") is the run-time procedure compiled by ." . Its purpose is to output the text string compiled into the dictionary by ." .

(.") performs two basic functions:

1. (.") prints the character string compiled into the dictionary by ." .
2. (.") increments the IP past the character string so that interpretation begins with the word following the string.

An example would be the printing of the characters "ABD" via the word ." (e.g., ." ABD"). The following would be compiled into the dictionary by ." .



(.") will first execute COUNT to set up the parameter stack for TYPE ; then it will increment the value on the return stack (which points to the next definition to be interpreted) to point to the "next definition's CFA".

* **At entry** - The top of the return stack contains the address of the length byte of the text string. (This address is normally that of the next word to be executed but the text string is compiled "in line" and therefore immediately follows the execution address of (.") .)
* **At exit** - The top of the return stack contains the address of the next word to interpret.

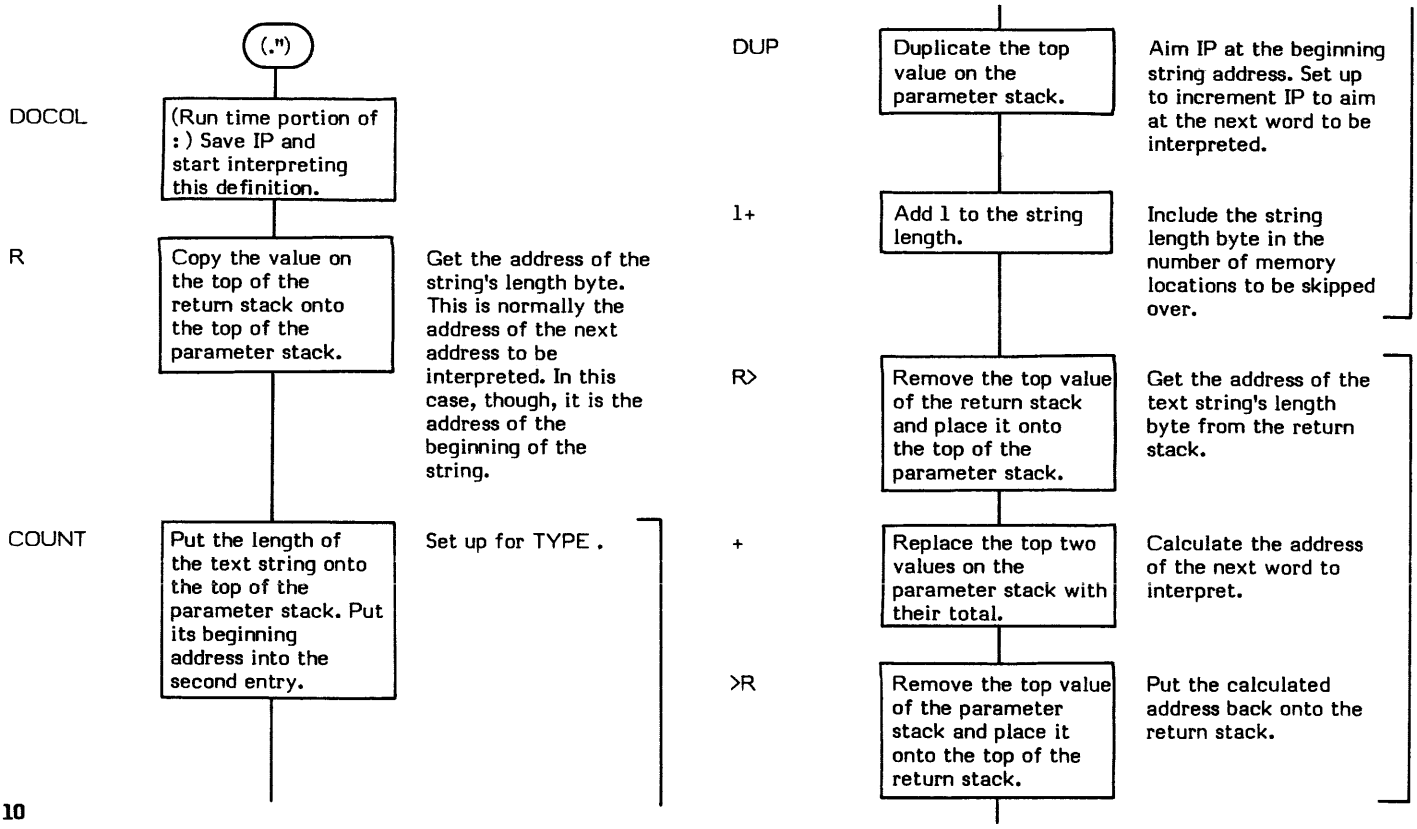
(.") is a high level colon definition.

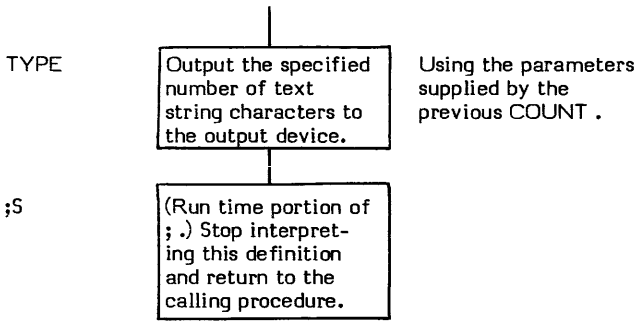
(.") may only be used inside of a colon definition.

Refer to ." .

FORTH-79: There is no FORTH-79 equivalent for (.") .

Definition: : (.") (—)
R COUNT DUP 1+ R> + >R TYPE ;




TYPE
Output the specified number of text string characters to the output device.
Using the parameters supplied by the previous COUNT .
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

(+ LOOP)

(+LOOP) (increment value --)

(+LOOP) (pronounced "paren-plus-loop") is the execution time (Sequence 3) procedure for +LOOP . During compile time (Sequence 2), +LOOP compiles the CFA of (+LOOP) into the definition being created.

The purpose of (+LOOP) is to serve as the run time end of a DO +LOOP sturcture. In doing so, it performs four primary functions:

1. (+LOOP) obtains the loop increment value from the top of the parameter stack.
2. (+LOOP) increments the loop Index by the increment value. Note that this value can be either positive or negative.
3. (+LOOP) performs a signed comparison of the newly calculated loop Index and the loop Limit.
4. (+LOOP) executes a branch back to the "DO" portion of the structure until the Index is either equal to or greater than the Limit when incrementing by a positive value or when the Index is greater than the Limit when incrementing by a negative value.

When either of these conditions occur, the Index and Limit are dropped from the return stack and execution continues with the word after the DO +LOOP .

(+LOOP) differs from (LOOP) only in that the increment value is provided on the top of the parameter stack instead of defaulting to a value of 1. Therefore, the code for (+LOOP) simply picks up the increment value from the parameter stack and jumps into the (LOOP) routine immediately after (LOOP) defaulted to an increment value of 1. (Note that this refers to the 8080 fig-FORTH Version 1.1 and differs from the fig-Model.)

Refer to +LOOP for a more high level description of the action of a DO +LOOP structure.

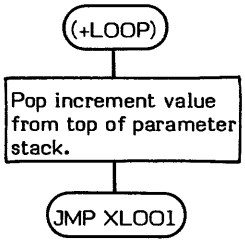
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision increment value. Note that this value may be negative. The top of the return stack contains the signed 16-bit Index value. The second return stack entry contains the 16-bit signed Limit value.
* **At exit to DO portion of loop** - No parameter stack parameters. The return stack contains the current Index value in the top entry and the original Limit value in the second entry.
* **At exit from DO +LOOP** - No parameter stack parameters. The Index and Limit values are dropped from the return stack.

(+LOOP) is a low level code primitive.

Refer to +LOOP , DO , and (LOOP) .

FORTH-79: There is no FORTH-79 equivalent for (+LOOP) .

8080 fig-FORTH
Version 1.1



Note this differs from the fig-Model.

Refer to (LOOP). Note this is an entry point in the code for (LOOP) as mentioned in description of (+LOOP) .

(;CODE) (pronounced "paren-semicolon-code") is a word normally used during Sequence 2 compilation. Its purpose is to compile the beginning address of the assembly language code (which must physically immediately follow (;CODE)) into the Code Field of the definition being compiled.

(;CODE) is actually the run time procedure for ;CODE . ;CODE is usually executed at Sequence 1 when defining a "parent" word. The "parent" defining word is then executed at Sequence 2 time to create "child" definitions.

Refer to ;CODE for a complete description of this "parent"/"child" relationship and the use of ;CODE .

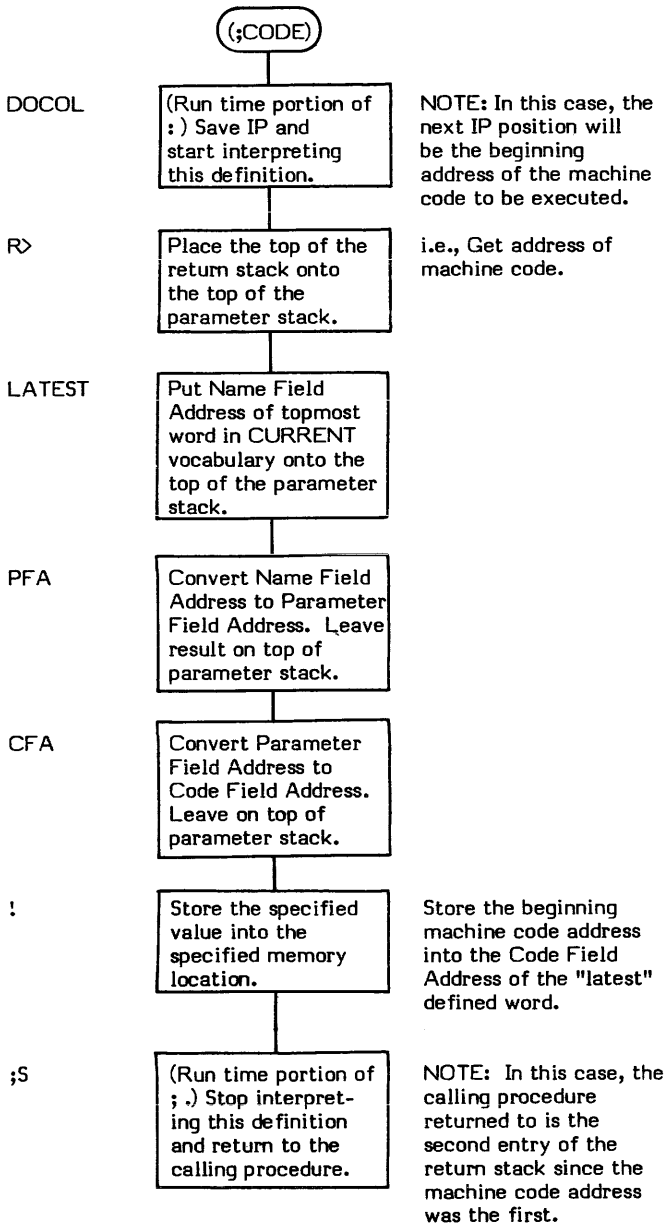
* **At entry** - No parameter stack parameters. The top of the return stack contains the address of the machine language code to be executed. The second word on the return stack contains the address of the procedure that control should be returned to after execution. Note this is the reason that definitions ending in ;CODE (assembly language) do not need to be ended with a ; .
* **At exit** - No parameter stack parameters. The two return stack parameters are dropped from the return stack.

(;CODE) is a high level colon definition.

Refer to ;CODE .

FORTH-79: There is no FORTH-79 equivalent for (;CODE) .

Definition: : (; CODE) (--)
R> LATEST PFA CFA ! ;



(ABORT)

(ABORT) (all stack values —)

(ABORT) (pronounced "paren-abort") is an intermediate word used between ERROR and ABORT . It is normally executed when WARNING is negative and simply performs an ABORT .

(ABORT) is intended to be the "hook" for application error routines. Replacing the CFA for ABORT with that of a user defined error routine and setting WARNING to -1 allows the user routine to "get control" when an error occurs.

* **At entry** - The parameter and return stacks may contain any values.
* **At exit (If an ABORT was executed)** - Both stacks will be cleared and the system will be set to the execution state. Refer to ABORT .
* **At exit (If an ABORT was not executed)** - Both stacks are unchanged and no action was taken.

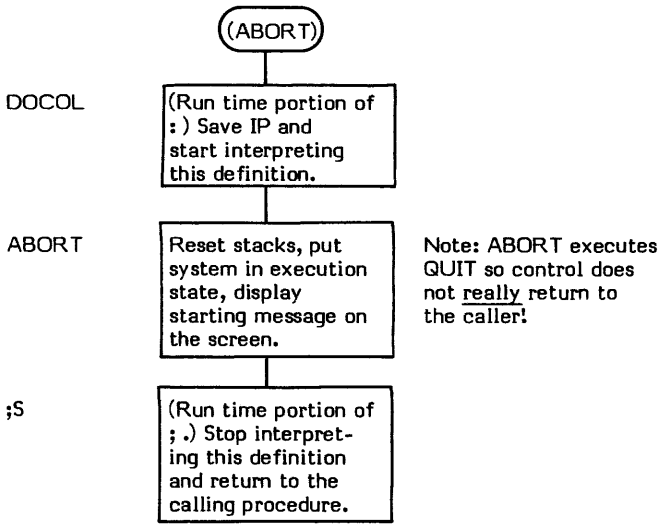
(ABORT) is a high level colon definition.

Refer to ABORT , WARNING , and ERROR .

FORTH-79: There is no FORTH-79 equivalent for (ABORT) .

Definition:

```
:  ( ABORT )  ( -- )
      ABORT  ;
```



(DO) **(Limit \ Index —)**

(DO) (pronounced "paren-do") is the execution time (Sequence 3) procedure for DO . During compile time (Sequence 2), DO compiles the CFA (Code Field Address) of (DO) into the definition being created.

The purpose of (DO) is to initialize a DO-LOOP structure for execution. It does this by transferring the user supplied Limit and Index values from the parameter stack to the return stack. The Limit and Index values can then be accessed by the run time procedures (LOOP) and (+LOOP) .

For example:

PARAM STACK	RETURN STACK	PARAM STACK	RETURN STACK
Index	n3	n1	Index
Limit	n4	n2	Limit
n1			n3
n2			n4
Before Execution Of (DO)		After Execution Of (DO)	

Note that the Index is the top value both on the parameter stack and on the return stack.

Also note that when the loop completes, (LOOP) or (+LOOP) have removed the Index and Limit from the return stack.

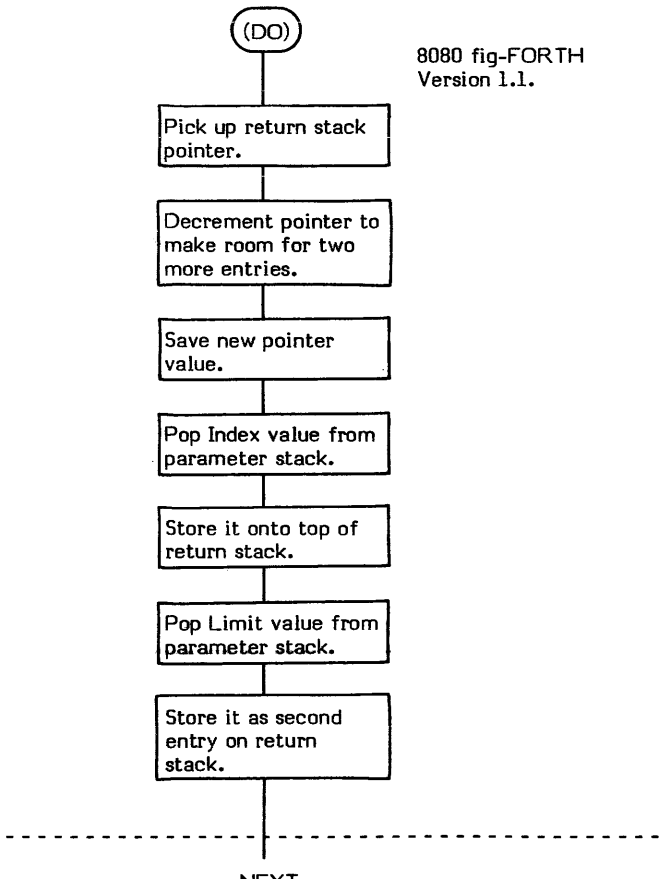
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision Index (or Initial) value. The second stack entry contains a signed 16-bit single precision Limit value. (Refer to DO .)

* **At exit** - The two parameters, Index and Limit, have been transferred to the return stack in the same order they appeared on the parameter stack.

(DO) is a low level code primitive.

Refer to DO , LOOP , +LOOP , (LOOP) , and (+LOOP) .

FORTH-79: There is no FORTH-79 equivalent for (DO) .



(FIND)

(FIND) Successful - (string addr\ NFA -- PFA\ length\ true flag)

(FIND) Unsuccessful - (string addr \ NFA -- false flag)

(FIND) (pronounced "paren-find") performs a dictionary search starting from a specified Name Field Address. (FIND) then looks for a match on the character string pointed to by the second stack word. (FIND) will search an entire "branch" of the vocabulary dictionary, "inwardly" toward the "trunk". The search stops when a 0 Link Field is encountered; usually, but not necessarily at the end of the FORTH vocabulary.

* **At entry** - The top of the parameter stack contains a 16-bit address pointer to the length byte (i.e., the first byte) of a Name Field in the dictionary. The second stack entry contains a pointer (NFA) to the character string to be used for comparison. The first byte of this string contains the length of the following string.

* **At exit - Successful** - The top of the parameter stack contains a boolean true flag (1). The second stack entry contains the byte length of the Name Field (with the MSB set denoting the length byte). The third stack entry contains a 16-bit address pointer (PFA) to the Parameter Field of the "found" dictionary entry.

* **At exit - Unsuccessful** - The top of the parameter stack contains a boolean false (0) flag.

(FIND) is a low level code primitive.

Refer to FIND , and VOCABULARY .

FORTH-79: There is no FORTH-79 equivalent for (FIND) .

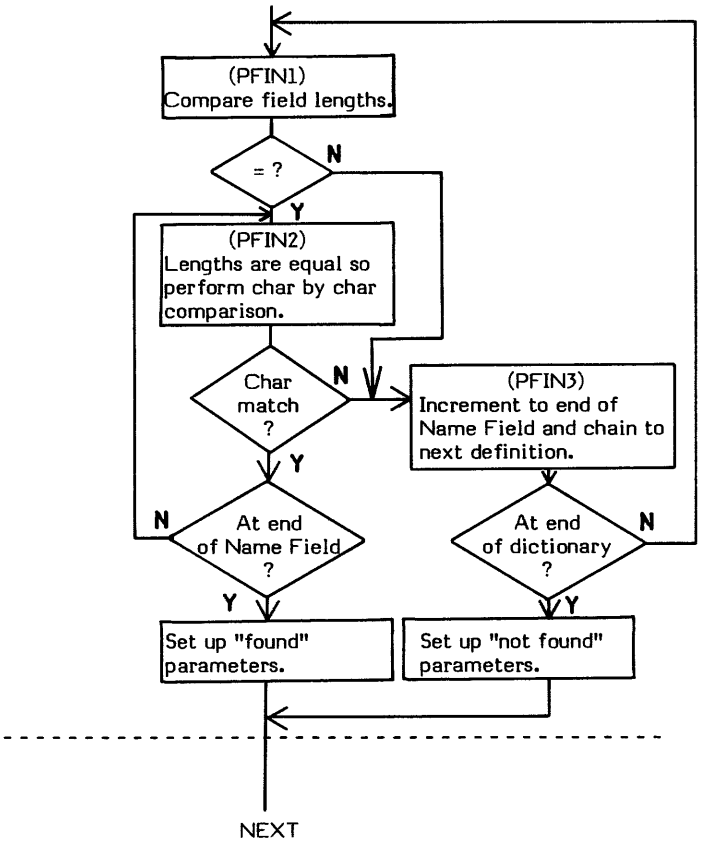
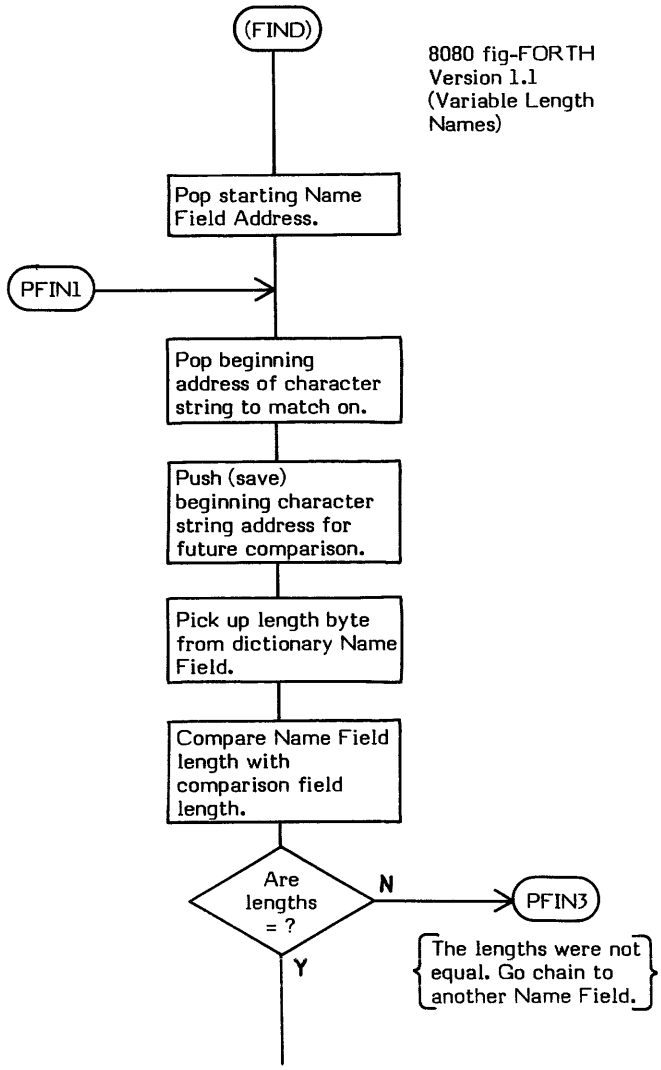
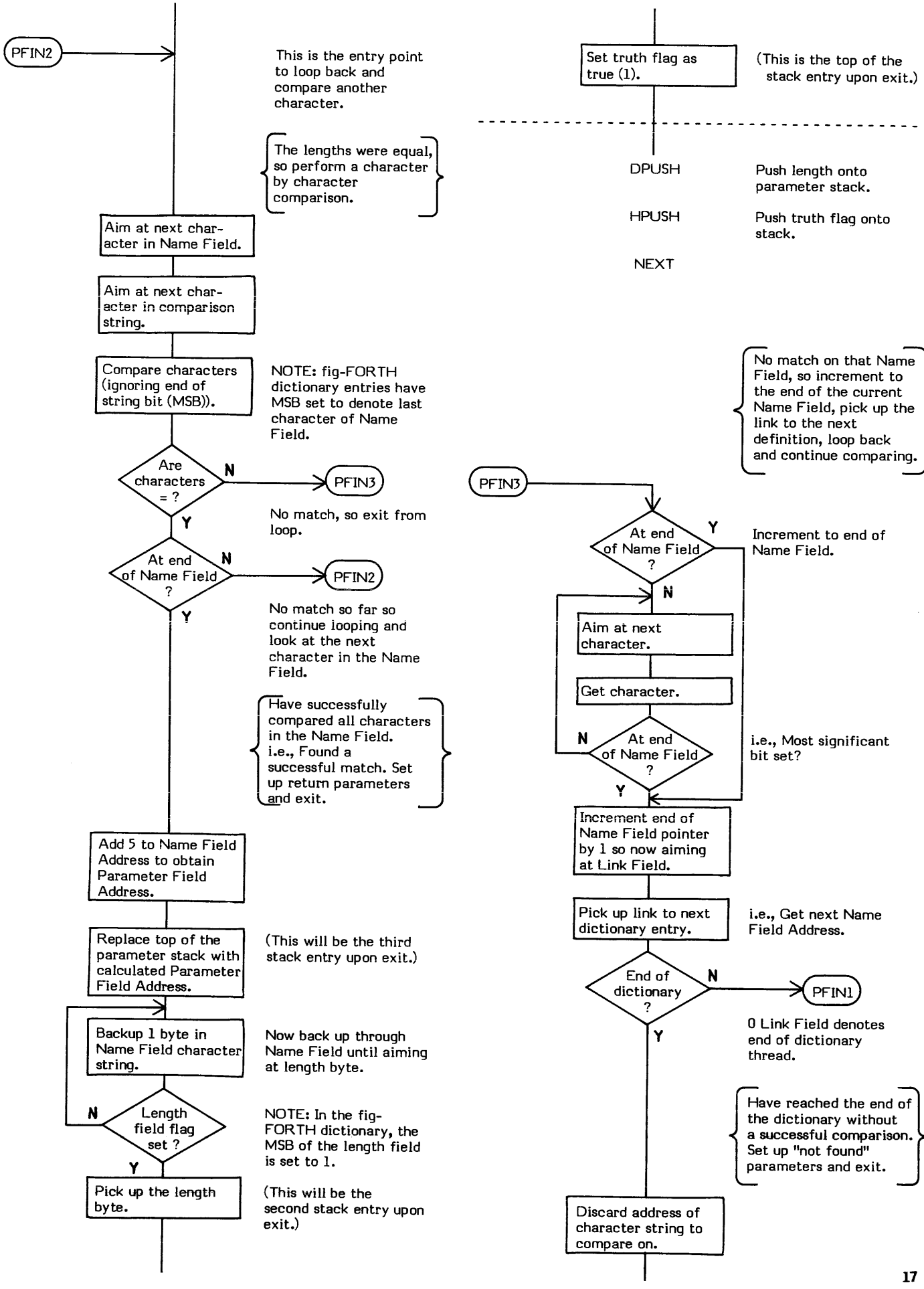
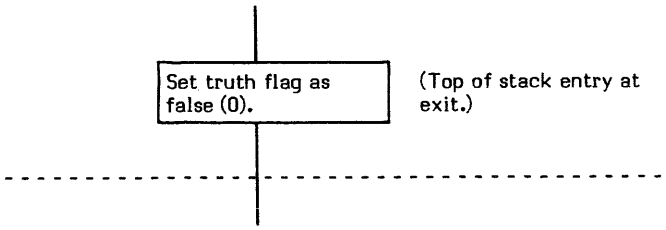


Figure (FIND)-1
High Level Flowchart of (FIND)

Each box roughly corresponds to the curly bracket comments in the low level flowchart.




PFIN2
This is the entry point to loop back and compare another character.
The lengths were equal, so perform a character by character comparison.
Aim at next character in Name Field.
Aim at next character in comparison string.
Compare characters (ignoring end of string bit (MSB)).
NOTE: fig-FORTH dictionary entries have MSB set to denote last character of Name Field.
Are characters = ?
N
PFIN3
No match, so exit from loop.
Y
At end of Name Field ?
N
PFIN2
No match so far so continue looping and look at the next character in the Name Field.
Y
Have successfully compared all characters in the Name Field. i.e., Found a successful match. Set up return parameters and exit.
Add 5 to Name Field Address to obtain Parameter Field Address.
Replace top of the parameter stack with calculated Parameter Field Address.
(This will be the third stack entry upon exit.)
Backup 1 byte in Name Field character string.
Now back up through Name Field until aiming at length byte.
N
Length field flag set ?
NOTE: In the fig-FORTH dictionary, the MSB of the length field is set to 1.
Y
Pick up the length byte.
(This will be the second stack entry upon exit.)
Set truth flag as true (1).
(This is the top of the stack entry upon exit.)
DPUSH
Push length onto parameter stack.
HPUSH
Push truth flag onto stack.
NEXT
No match on that Name Field, so increment to the end of the current Name Field, pick up the link to the next definition, loop back and continue comparing.
PFIN3
At end of Name Field ?
Y
Increment to end of Name Field.
N
Aim at next character.
Get character.
N
At end of Name Field ?
i.e., Most significant bit set?
Y
Increment end of Name Field pointer by 1 so now aiming at Link Field.
Pick up link to next dictionary entry.
i.e., Get next Name Field Address.
End of dictionary ?
N
PFIN1
0 Link Field denotes end of dictionary thread.
Y
Have reached the end of the dictionary without a successful comparison. Set up "not found" parameters and exit.
Discard address of character string to compare on.




```
HPUSH       Push truth flag onto
            parameter stack.

NEXT
```

(LINE) (line # \ screen # -- beg line addr \ line length)

(LINE) (pronounced "paren-line") converts a specified line number and screen number into the disk buffer address of the specified line. The value for the line length used is also returned. The specified screen is read into memory if necessary. (LINE) is used by .LINE to set up parameters for TYPE .

(LINE) performs three basic functions:

1. (LINE) calculates what block the line is in within the screen.
2. (LINE) then calculates the absolute block number.
3. (LINE) then executes BLOCK and calculates the memory address of the line.

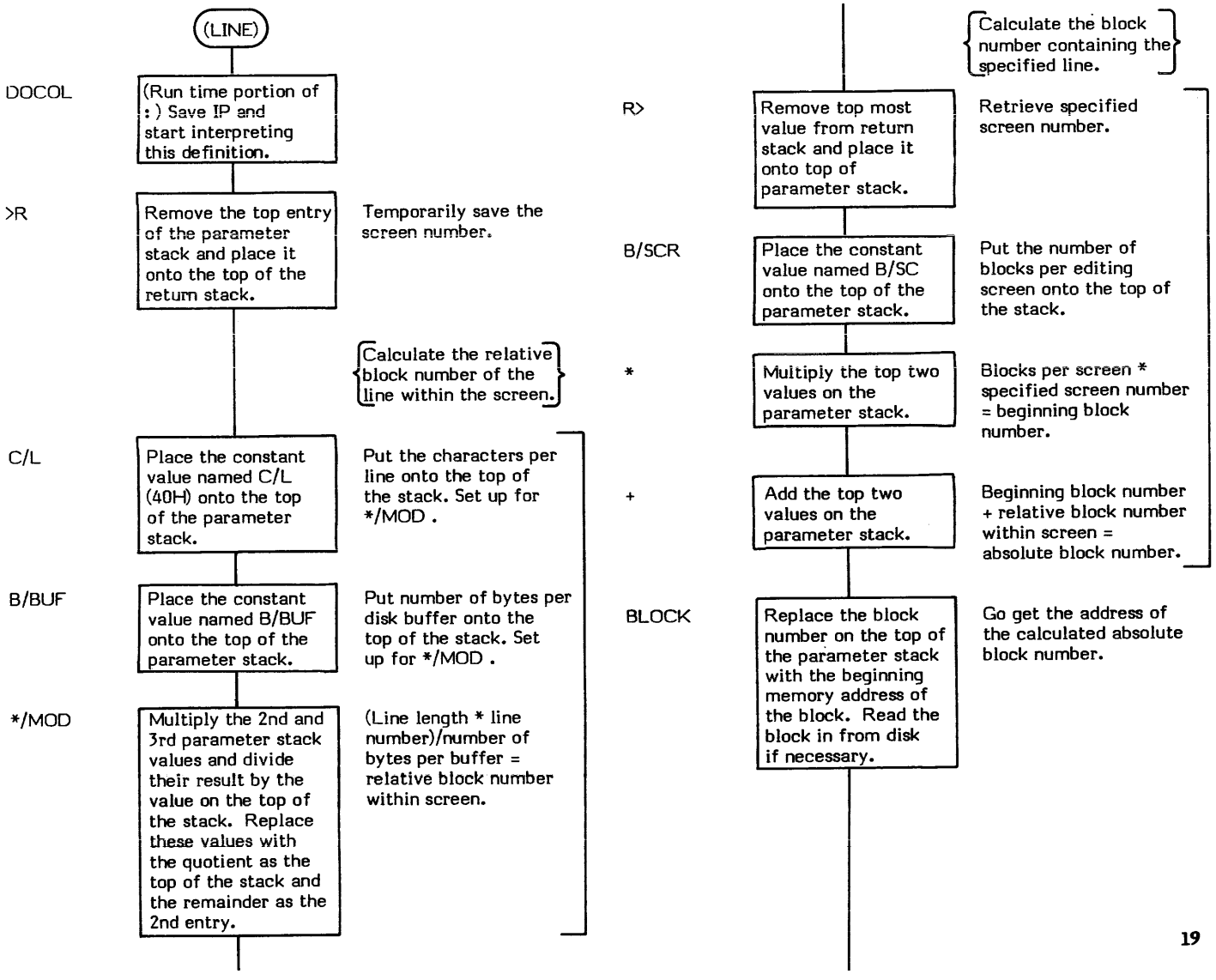
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value specifing the screen number to be used. The second stack entry contains a signed 16-bit single precision value specifying a line within the screen.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision byte count of the specified line. The second stack entry contains the 16-bit beginning memory address of the specified line. Trailing blanks are included. Use -TRAILING to suppress trailing blanks.

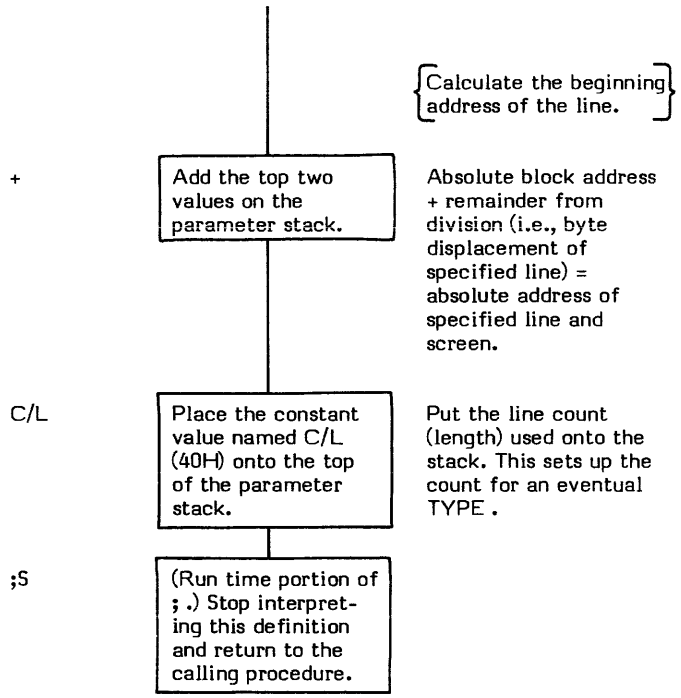
(LINE) is a high level colon definition.

Refer to .LINE .

FORTH-79: There is no FORTH-79 equivalent for (LINE) .

Definition:
```
:  (LINE)  ( line # \ screen # -- beg line addr \ line length )
        >R   C/L  B/BUF  */MOD   R>  B/SCR  *  +   BLOCK  +   C/L   ;
```




{Calculate the beginning address of the line.}
+
Add the top two values on the parameter stack.
Absolute block address + remainder from division (i.e., byte displacement of specified line) = absolute address of specified line and screen.
C/L
Place the constant value named C/L (40H) onto the top of the parameter stack.
Put the line count (length) used onto the stack. This sets up the count for an eventual TYPE .
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

(LOOP) (—)

(LOOP) (pronounced "paren-loop") is the execution time (Sequence 3) procedure for LOOP . During compile time (Sequence 2), LOOP compiles the CFA of (LOOP) into the definition being created.

The purpose of (LOOP) is to serve as the run time end of a DO-LOOP structure by performing the following functions:

1. (LOOP) increments the loop Index by 1.
2. (LOOP) performs a signed comparison on the newly calculated loop Index and the loop Limit.
3. (LOOP) executes a branch back to the "DO" portion of the structure until the Index is either equal to or greater than the Limit. When this condition occurs, the Index and Limit are dropped from the return stack and execution continues ahead.

(LOOP) only differs from (+LOOP) in that the increment value defaults to 1 in (LOOP) instead of the increment value being provided on the top of the stack as for (+LOOP) .

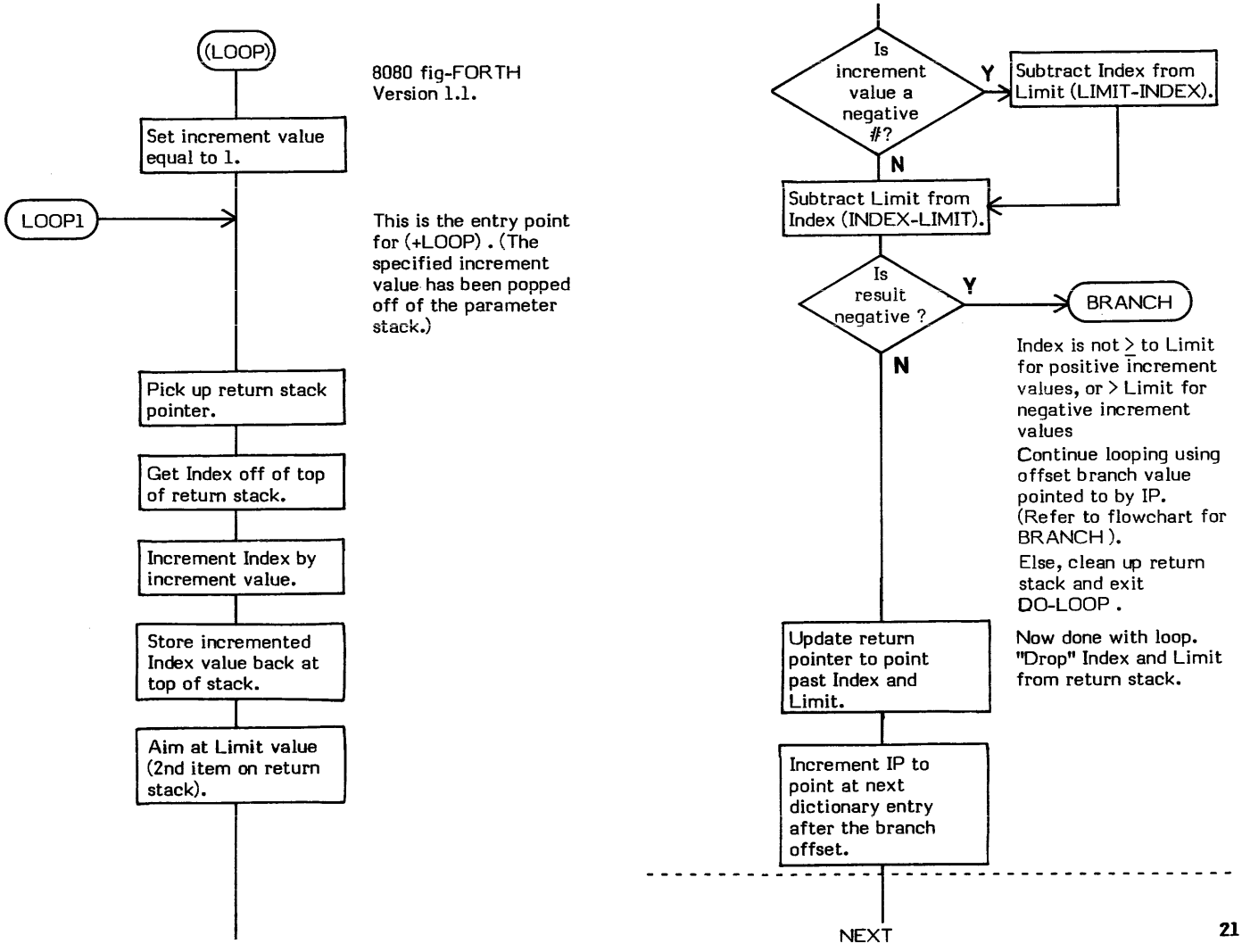
Refer to LOOP for a more high level description of the action of a DO-LOOP structure.

* **At entry** - No parameter stack entries. The top of the return stack contains the 16-bit signed Index value. The second return stack entry contains the 16-bit signed limit value.
* **At exit to DO portion of loop** - No parameter stack entries. The top of the return stack contains the 16-bit signed current Index value. The second return stack entry contains the 16-bit signed original Limit value.
* **At exit from DO-LOOP** - No parameter stack parameters. No return stack parameters.

(LOOP) is a low level code primitive.

Refer to LOOP , DO , I , and (+LOOP) .

FORTH-79: There is no FORTH-79 equivalent for (LOOP) .



(NUMBER)

(NUMBER) (double number \ string addr -- double number \ char addr)

(NUMBER) (pronounced "paren-number") converts a string of ascii text beginning at the specified address plus 1 (e.g., the length byte or decimal point is skipped over) into a double precision value of the radix specified in BASE . (NUMBER) is used primarily by NUMBER .

The user variable DPL is incremented to reflect the number of digits encountered to the right of the decimal point, provided that DPL has been set to a value other than -1 (refer to NUMBER).

* **At entry** - The top of the parameter stack contains the 16-bit address of the ascii character string to convert. The first byte of the character string is ignored.

  The second and third entries on the parameter stack contain a 32-bit double precision value (the most significant word being the second entry) into which the converted number is accumulated. This value should be 0 initially, but will be shifted left one digit in the radix when the next digit is converted.

* **At exit** - The top of the parameter stack contains a 16-bit address which points to the first non-convertible character encountered.

  The second and third stack entries contain the converted 32-bit unsigned double precision value. The second stack entry contains the high order portion of the value. The third stack entry contains the low order portion.

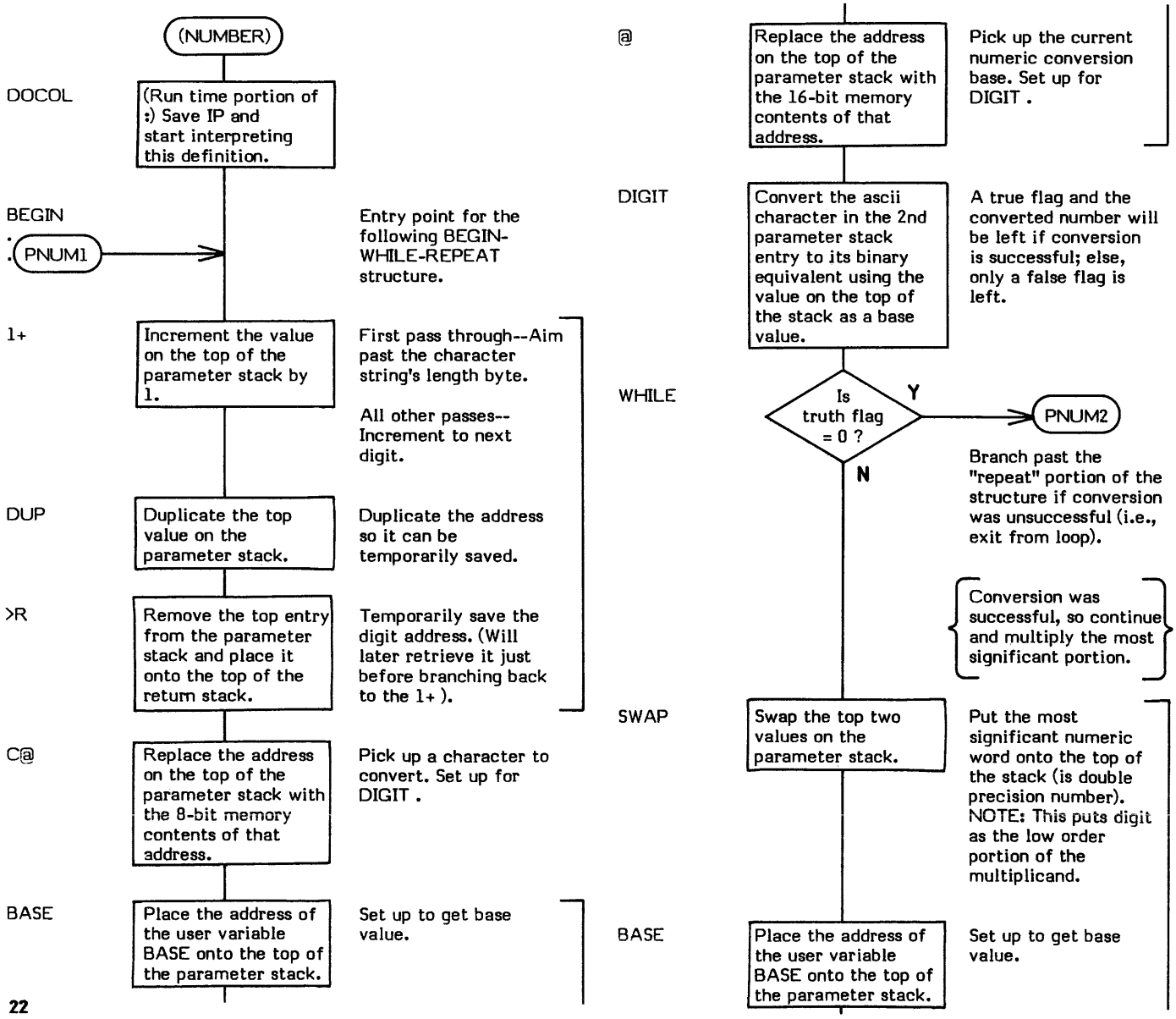
(NUMBER) is a high level colon definition.

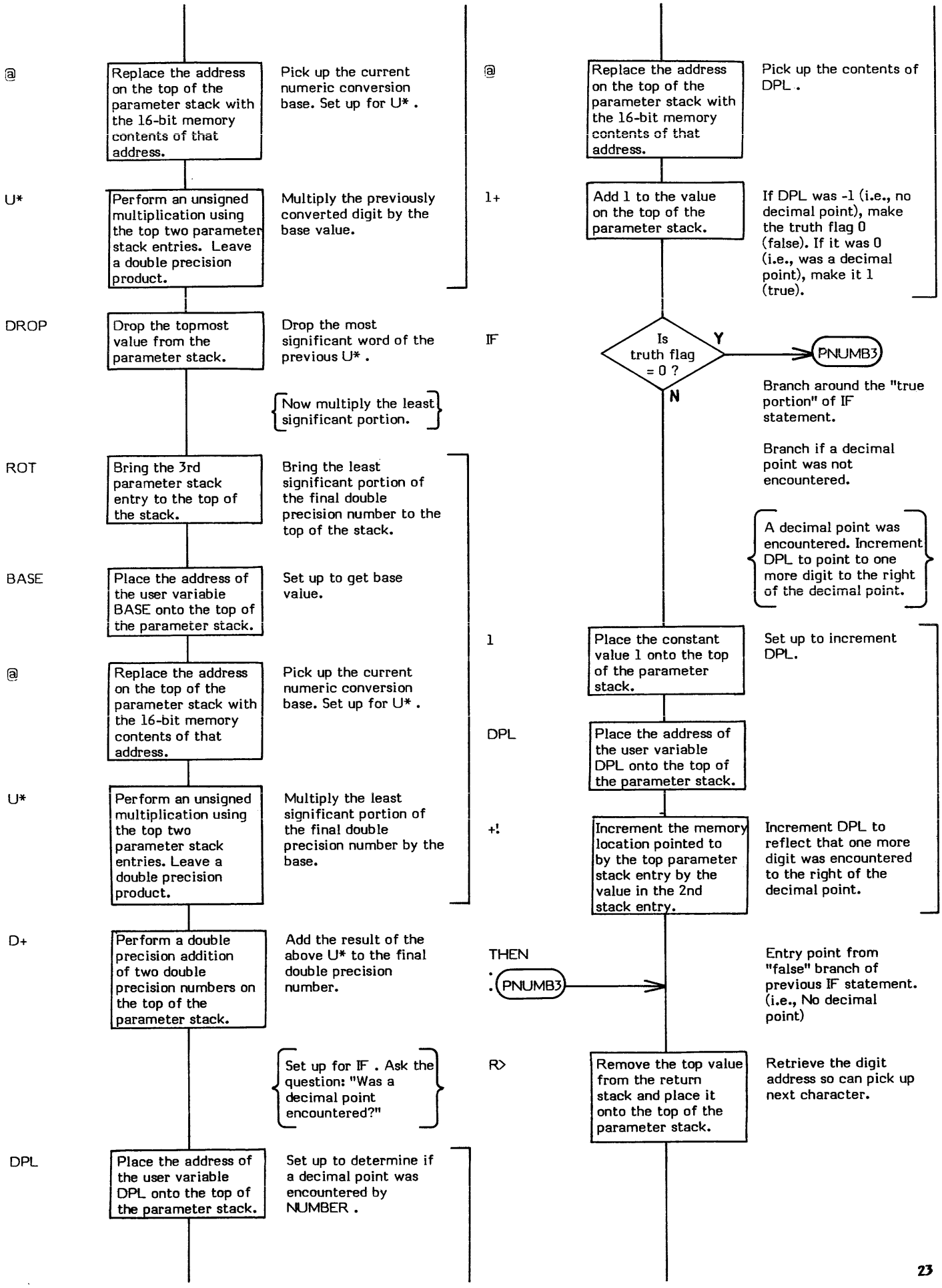
Refer to NUMBER .

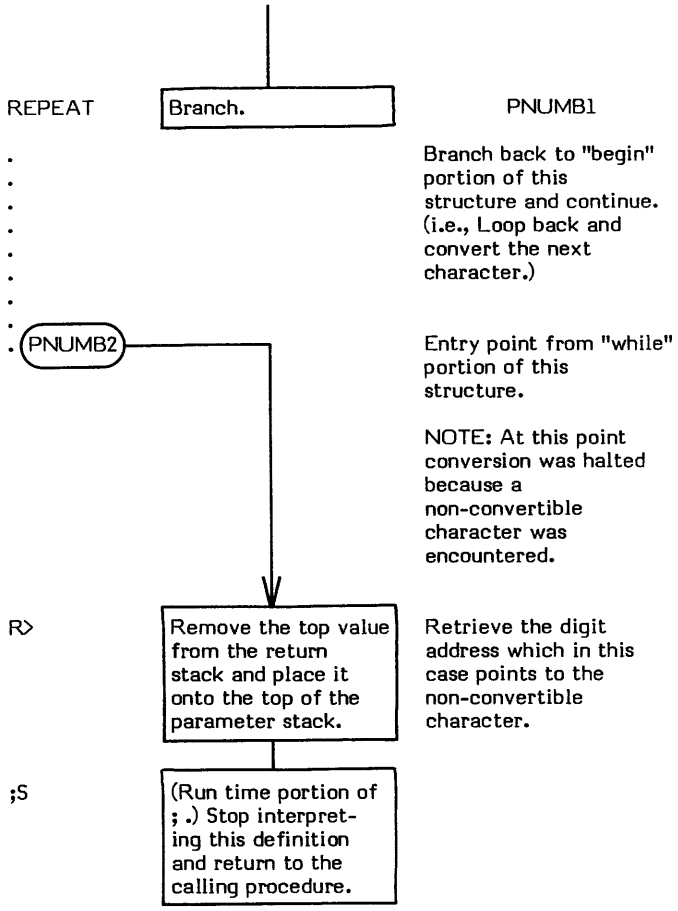
FORTH-79: The FORTH-79 equivalent for (NUMBER) is CONVERT .

Definition:

```
:  ( double number \ string addr -- double number \ char addr )
     BEGIN    1+ DUP >R    C@    BASE @    DIGIT
     WHILE    SWAP BASE @ U*    DROP    ROT BASE @ U*
              D+    DPL @ 1+    IF    1 DPL +!    THEN
              R>
     REPEAT   R>    ;
```




@
Replace the address on the top of the parameter stack with the 16-bit memory contents of that address.
Pick up the current numeric conversion base. Set up for U* .
U*
Perform an unsigned multiplication using the top two parameter stack entries. Leave a double precision product.
Multiply the previously converted digit by the base value.
DROP
Drop the topmost value from the parameter stack.
Drop the most significant word of the previous U* .
Now multiply the least significant portion.
ROT
Bring the 3rd parameter stack entry to the top of the stack.
Bring the least significant portion of the final double precision number to the top of the stack.
BASE
Place the address of the user variable BASE onto the top of the parameter stack.
Set up to get base value.
@
Replace the address on the top of the parameter stack with the 16-bit memory contents of that address.
Pick up the current numeric conversion base. Set up for U* .
U*
Perform an unsigned multiplication using the top two parameter stack entries. Leave a double precision product.
Multiply the least significant portion of the final double precision number by the base.
D+
Perform a double precision addition of two double precision numbers on the top of the parameter stack.
Add the result of the above U* to the final double precision number.
Set up for IF . Ask the question: "Was a decimal point encountered?"
DPL
Place the address of the user variable DPL onto the top of the parameter stack.
Set up to determine if a decimal point was encountered by NUMBER .
@
Replace the address on the top of the parameter stack with the 16-bit memory contents of that address.
Pick up the contents of DPL .
1+
Add 1 to the value on the top of the parameter stack.
If DPL was -1 (i.e., no decimal point), make the truth flag 0 (false). If it was 0 (i.e., was a decimal point), make it 1 (true).
IF
Is truth flag = 0 ?
Y
N
PNUMB3
Branch around the "true portion" of IF statement.
Branch if a decimal point was not encountered.
A decimal point was encountered. Increment DPL to point to one more digit to the right of the decimal point.
1
Place the constant value 1 onto the top of the parameter stack.
Set up to increment DPL.
DPL
Place the address of the user variable DPL onto the top of the parameter stack.
+!
Increment the memory location pointed to by the top parameter stack entry by the value in the 2nd stack entry.
Increment DPL to reflect that one more digit was encountered to the right of the decimal point.
THEN
PNUMB3
Entry point from "false" branch of previous IF statement. (i.e., No decimal point)
R>
Remove the top value from the return stack and place it onto the top of the parameter stack.
Retrieve the digit address so can pick up next character.


REPEAT
Branch.
PNUMB1
Branch back to "begin" portion of this structure and continue. (i.e., Loop back and convert the next character.)
PNUMB2
Entry point from "while" portion of this structure.
NOTE: At this point conversion was halted because a non-convertible character was encountered.
R>
Remove the top value from the return stack and place it onto the top of the parameter stack.
Retrieve the digit address which in this case points to the non-convertible character.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

* (value1 \ value2 -- single product)

* (pronounced "times") multiplies two signed single precision values and replaces them with their signed single precision product.

* is simply a M* followed by a DROP to make the result a single precision value. (Note this differs from the fig-Model. The model uses U* which is an unsigned multiply.)

- **At entry** - The top of the parameter stack contains a signed 16-bit value to be multiplied by the signed 16-bit value in the second stack entry.
- **At exit** - The top of the parameter stack contains the signed 16-bit single precision product of the input multipliers.

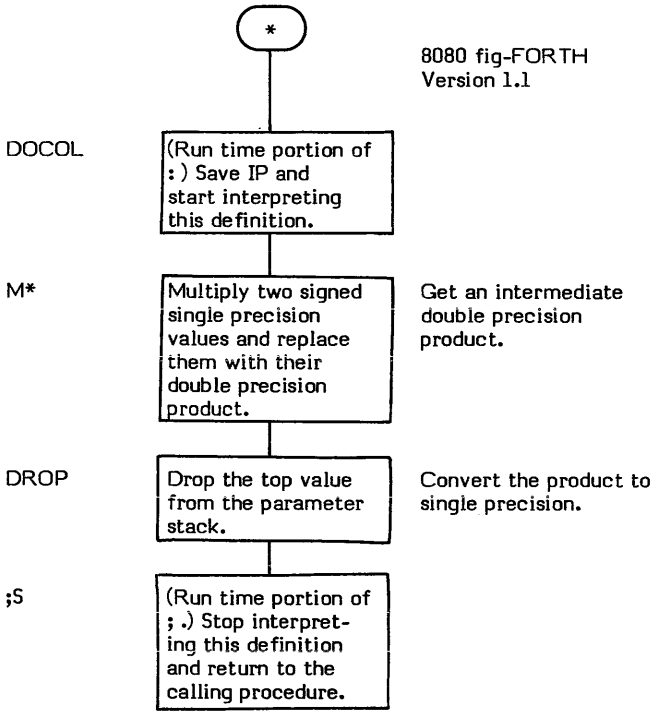
* is a high level colon definition.

Refer to M* .

FORTH-79: The FORTH-79 equivalent for * is * .

Definition:

```
:  *  ( value1 \value2 -- single product )
      M* DROP  ;
```



*/

***/ (multiplier \ multiplicand \ divisor -- quotient)**

*/ (pronounced "times-divide") performs a multiplication and then a division of three 16-bit signed single precision values according to the algebraic statement (value1 * value2) / value3. A 16-bit signed single precision quotient is the output of this operation.

Logically */ is the same as the sequence:

value1 value2 * value3 /

but */ carries the result of the multiplication as a 32-bit signed double precision intermediate result. This allows greater accuracy than if a single precision intermediate product was used.

The basis of */ is */MOD . */ drops the remainder generated by */MOD . If a remainder is desired, use */MOD .

* **At entry** - The parameter stack contains three 16-bit signed single precision values. The second and third stack entries are to be multiplied together with the resulting product divided by the 16-bit signed single precision value on the top of the parameter stack.

* **At exit** - The top of the parameter stack contains a 16-bit signed single precision quotient.

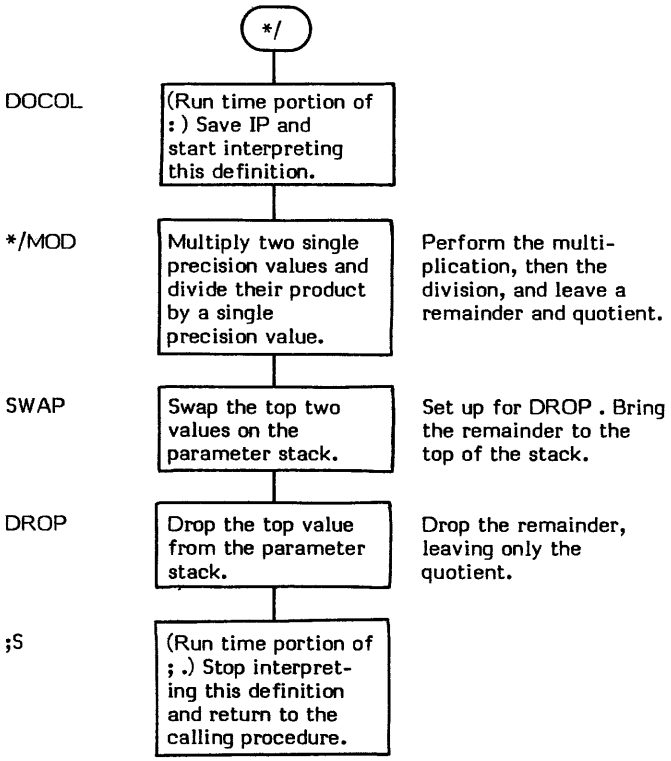
*/ is a high level colon definition.

Refer to */MOD .

FORTH-79: There is no FORTH-79 equivalent for */ .

Definition:

```
:  */  ( multiplier\multiplicand\divisor -- quotient )
       */MOD SWAP DROP  ;
```



***/MOD (multiplicand \ multiplier \ divisor -- remainder \ quotient)**

*/MOD (pronounced "times-divide-mod") performs a multiplication and then a division of three 16-bit signed single precision values according to the algebraic statement (value1 * value2) / value3.

Both a 16-bit signed single precision quotient and a 16-bit signed single precision remainder (hence the "MOD" in the name) are the output of this operation.

Logically */MOD is the same as the sequence:

```
value1  value2  *  value3  /
```

but */MOD carries the result of the multiplication as a 32-bit signed double precision intermediate result. This allows greater accuracy than if a single precision intermediate product was used.

*/MOD is arithmetically identical to */ except */ drops the remainder.

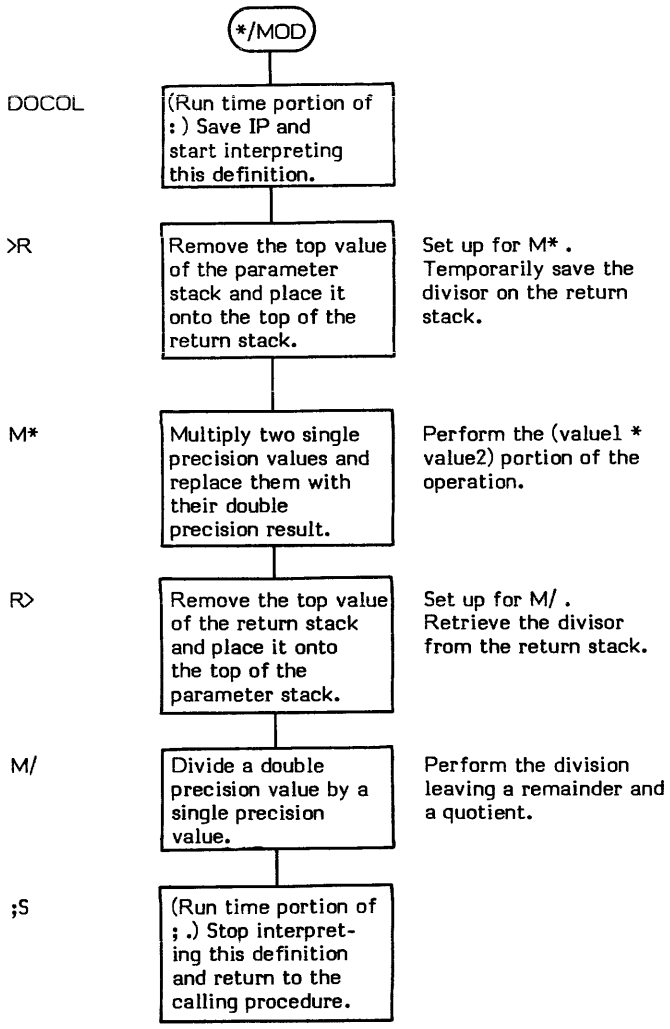
- **At entry** - The parameter stack contains three 16-bit signed single precision values. The second and third stack entries are to be multiplied together with the resulting product divided by the 16-bit signed single precision value on the top of the parameter stack.
- **At exit** - The top of the parameter stack contains the 16-bit signed single precision quotient. The second stack entry contains a 16-bit signed single precision remainder.

*/MOD is a high level colon definition.

Refer to */ .

FORTH-79: The FORTH-79 equivalent for */MOD is */MOD .

Definition:
```
:  */MOD  ( multiplicand\ multiplier\ division -- remainder\ quotient )
       >R  M*  R>  M/  ;
```



+

+ (value1 \ value2 — value)

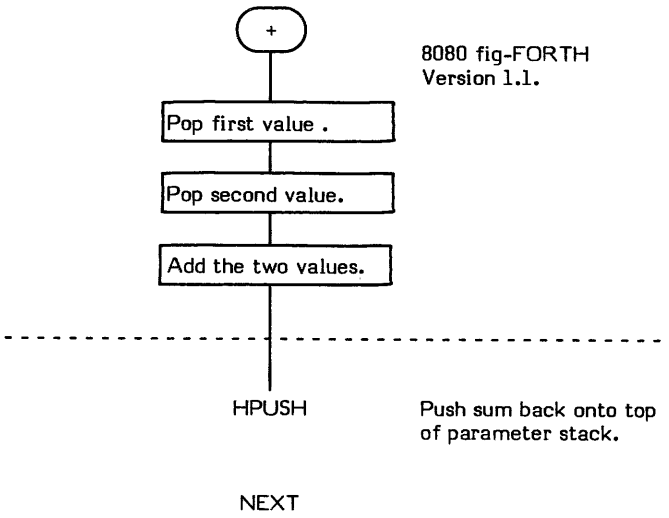
+ (pronounced "plus") adds the top two 16-bit signed numbers on the parameter stack and replaces them with their 16-bit signed sum. Note that generation of a carry goes unnoticed.

* **At entry** - The top and second entries of the parameter stack contain the signed 16-bit single precision values to be added.

* **At exit** - The top of the parameter stack contains the signed 16-bit single precision sum of the two values.

+ is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for + is + .



+! (value \address —)

+! (pronounced "plus-store") adds the 16-bit value contained in the second parameter stack entry to the 16-bit memory word addressed via the entry on the top of the stack.

This is a commonly used method of incrementing counters kept in memory.

HOLD is an example of a word that uses +! .

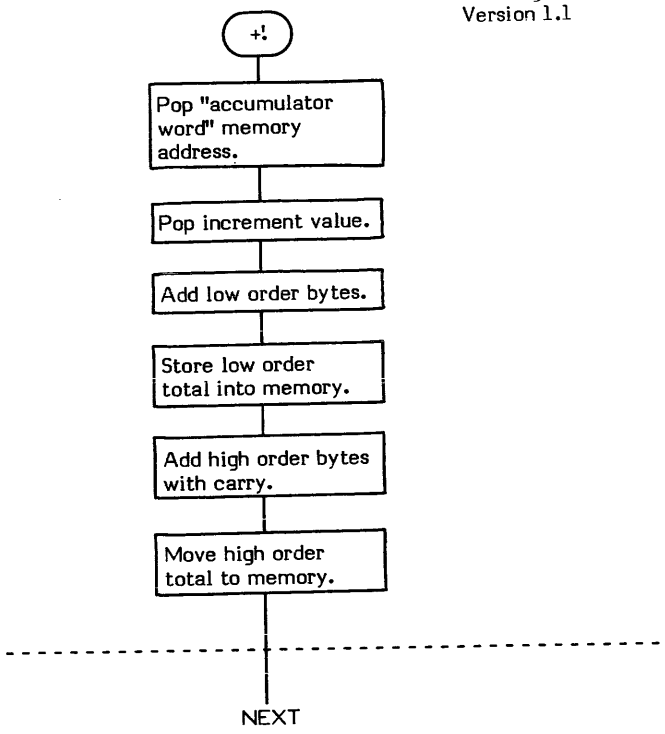
* **At entry** - The top of the parameter stack contains the address of a 16-bit memory word to be used as an accumulator. The second stack entry contains the signed 16-bit single precision value to be added to the word in memory.

* **At exit** - No parameters. The word in memory contains the sum of its previous contents and the specified value.

+! is a low level code primitive.

FORTH-79: The FORTH-79 equivalent of +! is +! .

8080 fig-FORTH
Version 1.1



+ -

+- (value to have sign set \ value whose sign is used -- value whose sign is set)

+- (pronounced "plus-minus") negates the sign of the second stack value if the sign of the top stack value is negative. The top value is then dropped.

The following truth table describes the outcome of all possible combinations:

Second Entry	Top of Stack	Results
+ V2	+ V1	+ V2
+ V2	- V1	- V2
- V2	+ V1	- V2
- V2	- V1	+ V2

ABS is an example of a word that uses +- .

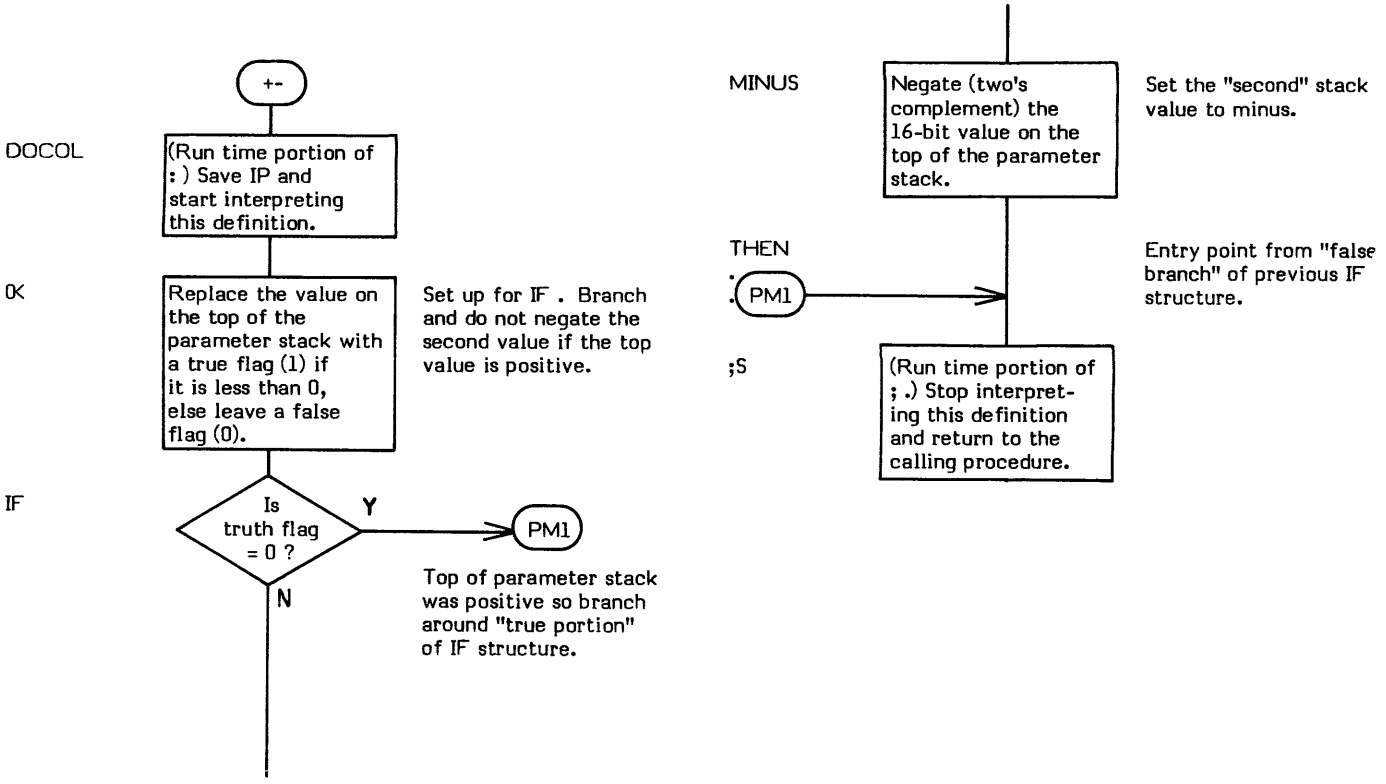
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value. The second stack entry contains the 16-bit signed single precision value which is to have its sign bit set.
* **At exit** - The top of the parameter stack contains the value originally in the second stack entry. The sign of this 16-bit signed value is set according to the truth table. The original top of stack value is dropped.

+- is a high level colon definition.

Refer to ABS , and D+- .

FORTH-79: There is no FORTH-79 equivalent for +- .

Definition: : +- (value\ signed value -- signed value)
0< IF MINUS THEN ;



+BUF (buffer address -- next buffer address \ flag)

+BUF (pronounced "plus-buff") advances the specified buffer address to the address of the next buffer.

The action of +BUF is directly related to the physical arrangement of the buffers in a FORTH system.

A system normally contains several buffers. (Refer to Figure +BUF-1.) Each buffer consists of a one word header, a data portion, and a terminator word. The header contains the block number and an "update" flag in the high order bit (see UPDATE). The length of the data body is specified by the constant B/BUF . The terminator word consists of nulls and is used to flag the interpreter that the end of the block has been reached.

These buffers are located in memory as a physically contiguous buffer array. While the buffers are physically contiguous, they are logically treated as a "circular" array. A reference to +BUF always returns the address of the next buffer in the circular array.

* **At entry** - The top of the parameter stack contains the 16-bit address of the beginning of a buffer.
* **At exit** - The top of the parameter stack contains a boolean flag which is false (0) if the returned buffer address equals that stored in PREV . The flag is true (not 0) if the returned address does not equal that in PREV . (This flag is primarily used by BLOCK to determine when all buffers have been scanned for the desired block number.) The second stack entry contains the 16-bit address of the next buffer in the "logically circular" buffer array.

+BUF is a high level colon definition.

Refer to BLOCK , PREV , UPDATE , and BUFFER .

FORTH-79: There is no FORTH-79 equivalent for +BUF .

Definition:

```
:  +BUF  ( buffer address -- next buffer address \flag )
     B/BUF 4 +  +  DUP  LIMIT =  IF  DROP FIRST  THEN
     DUP PREV @ -  ;
```

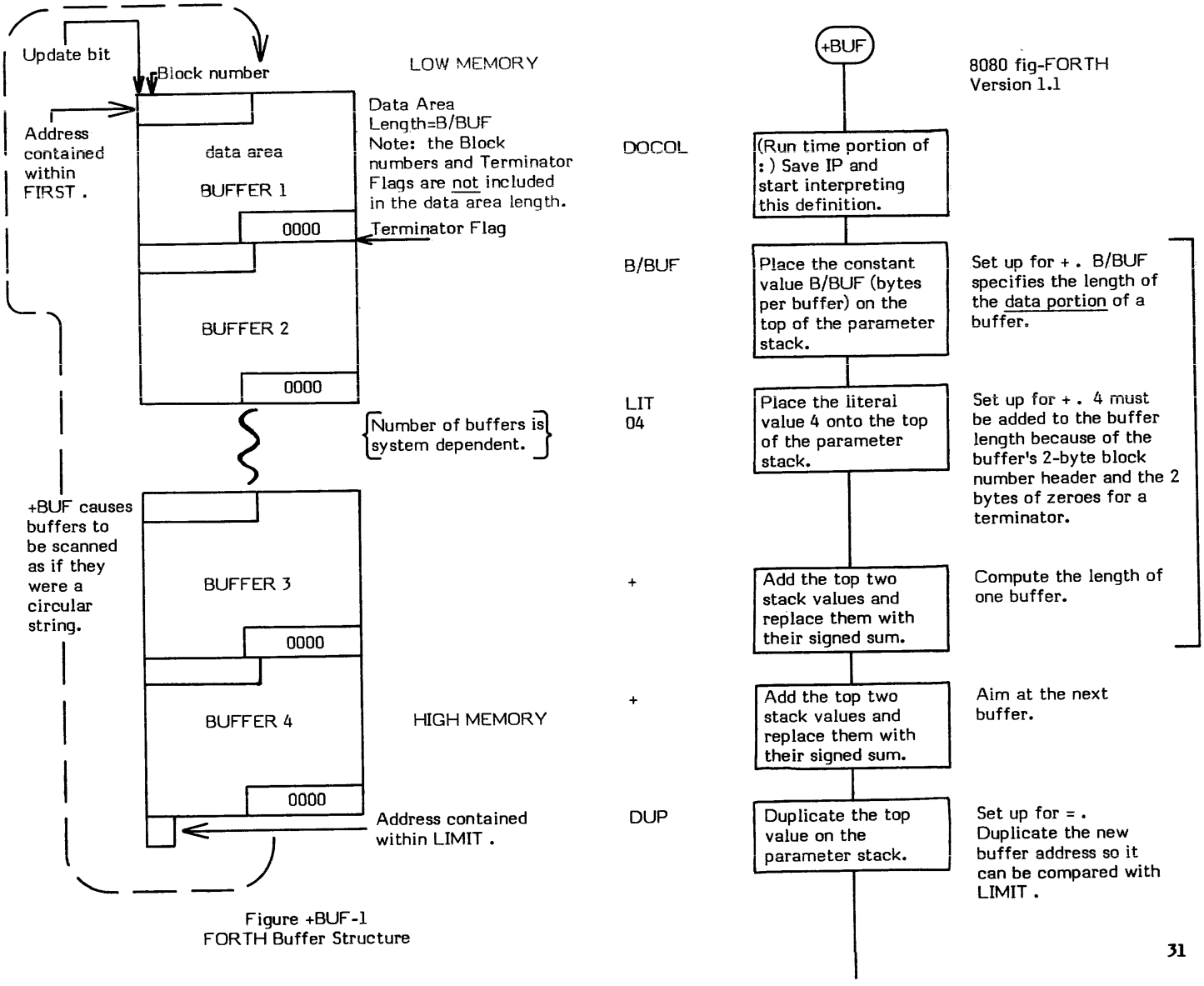
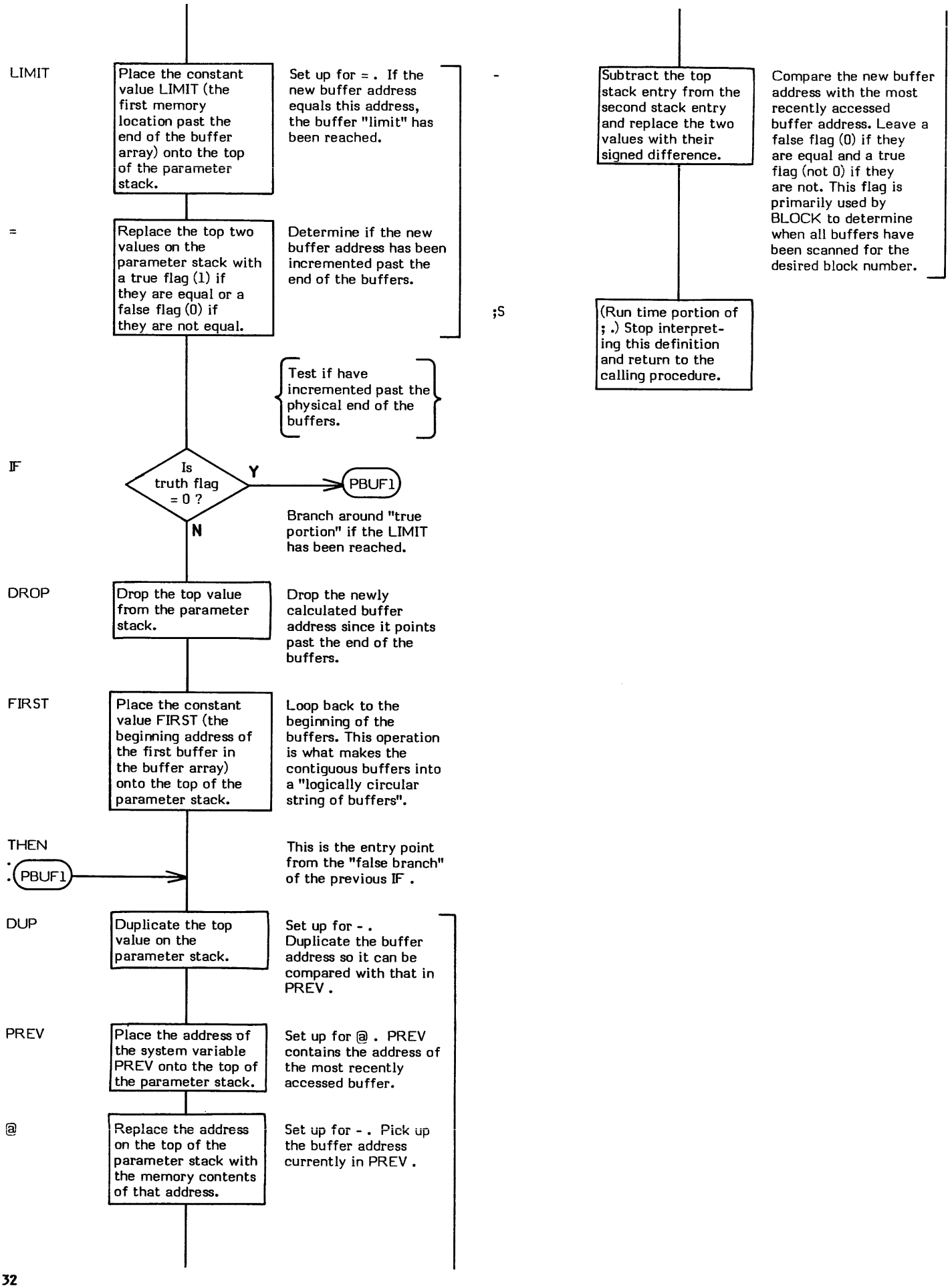


Figure +BUF-1
FORTH Buffer Structure


LIMIT
Place the constant value LIMIT (the first memory location past the end of the buffer array) onto the top of the parameter stack.
Set up for = . If the new buffer address equals this address, the buffer "limit" has been reached.
=
Replace the top two values on the parameter stack with a true flag (1) if they are equal or a false flag (0) if they are not equal.
Determine if the new buffer address has been incremented past the end of the buffers.
Test if have incremented past the physical end of the buffers.
IF
Is truth flag = 0 ?
Y
PBUF1
N
Branch around "true portion" if the LIMIT has been reached.
DROP
Drop the top value from the parameter stack.
Drop the newly calculated buffer address since it points past the end of the buffers.
FIRST
Place the constant value FIRST (the beginning address of the first buffer in the buffer array) onto the top of the parameter stack.
Loop back to the beginning of the buffers. This operation is what makes the contiguous buffers into a "logically circular string of buffers".
THEN
PBUF1
This is the entry point from the "false branch" of the previous IF .
DUP
Duplicate the top value on the parameter stack.
Set up for - . Duplicate the buffer address so it can be compared with that in PREV .
PREV
Place the address of the system variable PREV onto the top of the parameter stack.
Set up for @ . PREV contains the address of the most recently accessed buffer.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for - . Pick up the buffer address currently in PREV .
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Compare the new buffer address with the most recently accessed buffer address. Leave a false flag (0) if they are equal and a true flag (not 0) if they are not. This flag is primarily used by BLOCK to determine when all buffers have been scanned for the desired block number.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

+LOOP

COMPILE TIME: (loop address \ security check value —)
(Sequence 2)

EXECUTION TIME: (increment value —)
(Sequence 3)

+LOOP (pronounced "plus-loop") is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

+LOOP is used to end a DO-LOOP structure in conjunction with the word DO .

+LOOP may only be used within a colon-definition. Since +LOOP must be paired with a DO, DO leaves a 3 on the stack at compile time (Sequence 2) and +LOOP checks for this value. Since no other conditional or looping words leave a 3 on the stack, failure to pair +LOOP with a DO will cause an error condition to be signalled by ?PAIRS .

The apparent run time action (Sequence 3) of +LOOP is to increment the loop Index by the specified value and compare it with the Limit value. This action is actually performed by (+LOOP) . If the increment value is positive, a branch back to the "loop body" of the loop structure is executed if the Index is either larger than or equal to the Limit. If the increment value is negative, a branch back to the "loop body" is executed if the Index is larger than the Limit. If a "loop" back is not executed, the Index and Limit are dropped from the return stack and execution continues with the definition following +LOOP .

Note that +LOOP is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains a 16-bit single precision value used to provide compiler security. The value 3 is left on the stack at compile time by DO . The second stack entry contains a 16-bit address specifying the entry point of the DO portion (i.e., the beginning of the "loop body") of the DO-LOOP structure.

* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - (+LOOP) expects the top of the parameter stack to contain a 16-bit signed single precision increment value which is added to the Index value. The Index and Limit values are expected to be on the return stack. Refer to (+LOOP) .

* **At exit** - (+LOOP) drops the increment value from the parameter stack each time it is executed. It also drops the Index and Limit values from the return stack when the DO-LOOP structure is exited.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

+LOOP is a high level colon definition.

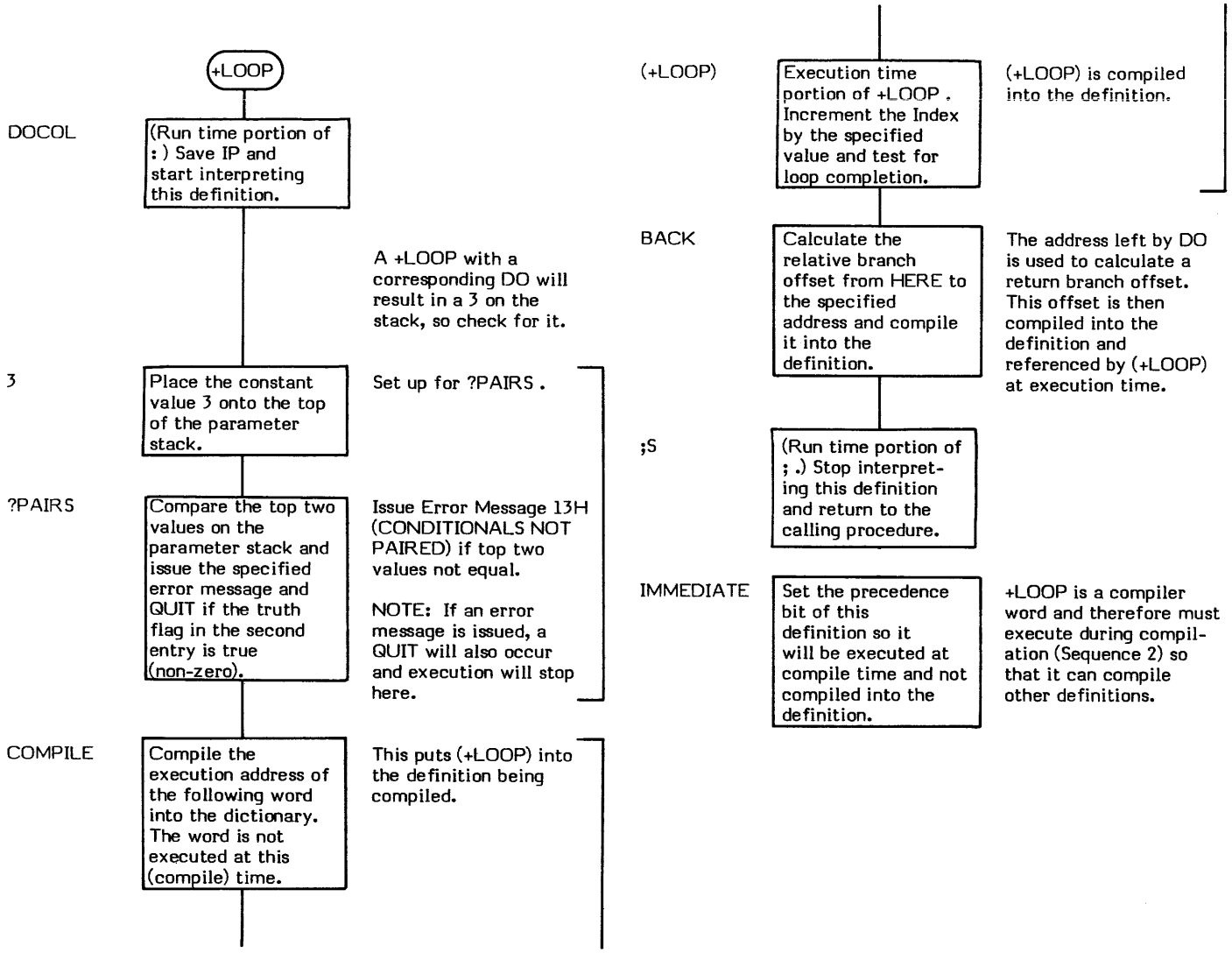
Refer to (+LOOP) , DO , LOOP , and (DO) .

FORTH-79: The FORTH-79 equivalent for +LOOP is +LOOP .

Definition:

```
:  +LOOP  ( loop address \ security check value ) ( compile time parameters )
       3 ?PAIRS  COMPILE (+LOOP)  BACK  ;  IMMEDIATE
```

COMPILE TIME action of +LOOP (Sequence 2): (loop address \ security check value —)



EXECUTION TIME action of +LOOP (Sequence 3): (increment value —)

Refer to (+LOOP) for the RUN TIME action of +LOOP .

+ORIGIN

COMPILE TIME (Sequence 2): (—)

EXECUTION TIME (Sequence 3): (offset — address)

+ORIGIN (pronounced "plus-origin") is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

+ORIGIN is used to obtain the address of a particular start-up parameter located within the "origin parameters".

The "ORIGIN" is a location in the FORTH system that marks the beginning of a series of parameters primarily used to initialize the system (see COLD).

The following list of parameters, taken from the 8080 fig-FORTH Version 1.1 listing, are used to initialize the user variables:

OFFSET	PARAMETER	DESCRIPTION
0	S0	Initial parameter stack pointer address.
2	R0	Initial return stack pointer address.
4	TIB	Terminal Input Buffer address.
6	WIDTH	The number of characters saved in a Name Field.
8	WARNING	The flag denoting error message status.
10	FENCE	The address below which the dictionary cannot be forgotten.
12	DP	The next available dictionary location.
14	VOC-LINK	The pointer to the last chronologically added vocabulary (FORTH at start up).

By supplying +ORIGIN with a byte offset (or word offset on word addressable machines), the address of a specific parameter can be obtained.

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains a signed (best to be positive though) single precision offset value.
* **At exit** - The top of the parameter stack contains the 16-bit address of the specified origin parameter.

+ORIGIN is a high level colon definition.

Refer to COLD and the user variables S0 , R0 , TIB , WIDTH , WARNING , FENCE , DP , and VOC-LINK .

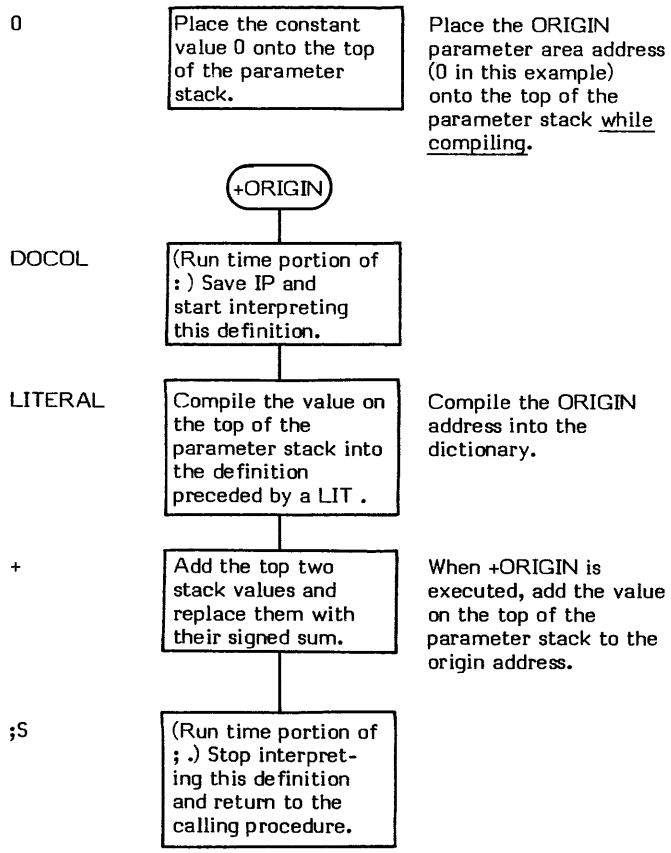
FORTH-79: There is no FORTH-79 equivalent for +ORIGIN .

Definition:

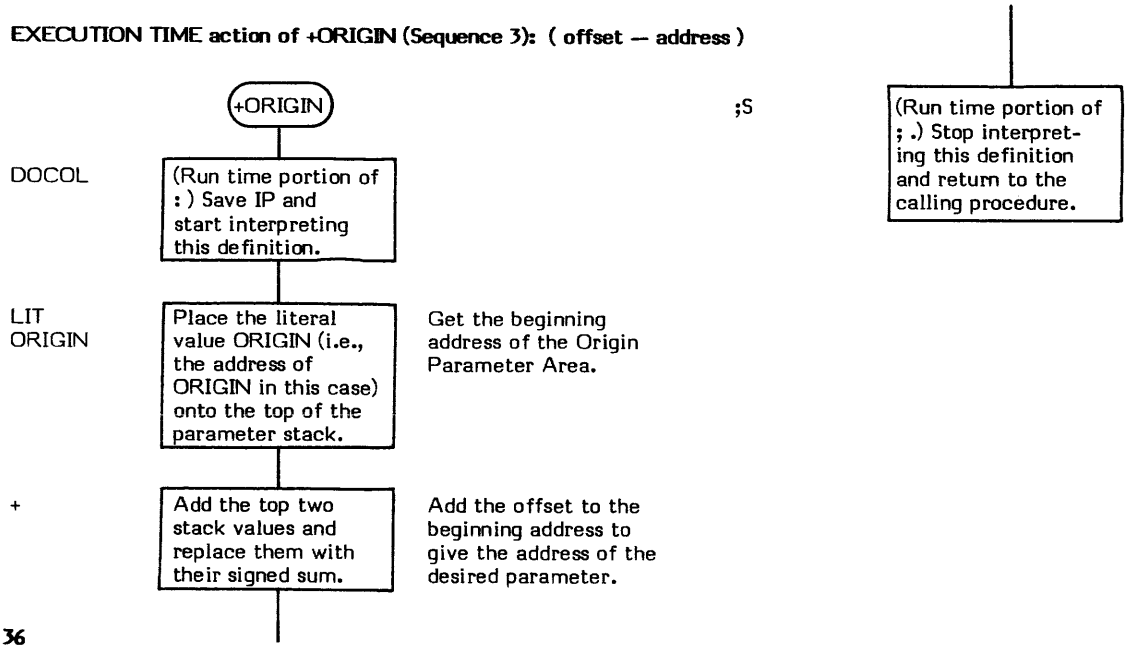
```
:  +ORIGIN  ( offset -- address ) ( execution time )
       ORIGIN +  ;
```

COMPILE TIME action of +ORIGIN (Sequence 2): (—)

This compile time action of +ORIGIN is shown because of the odd way the FORTH fig-Model provides the origin area address at compilation time.



EXECUTION TIME action of +ORIGIN (Sequence 3): (offset — address)



, (value —)

, (pronounced "comma") compiles (i.e., stores) the 16-bit value on the top of the parameter stack into the next available dictionary location and advances the dictionary pointer.

, is the primary compiler word. This is the principle means of compiling a word into the dictionary (e.g., INTERPRET uses , to compile definitions into the dictionary).

* **At entry** - The top of the parameter stack contains the value to be stored into the dictionary.
* **At exit** - No parameters.

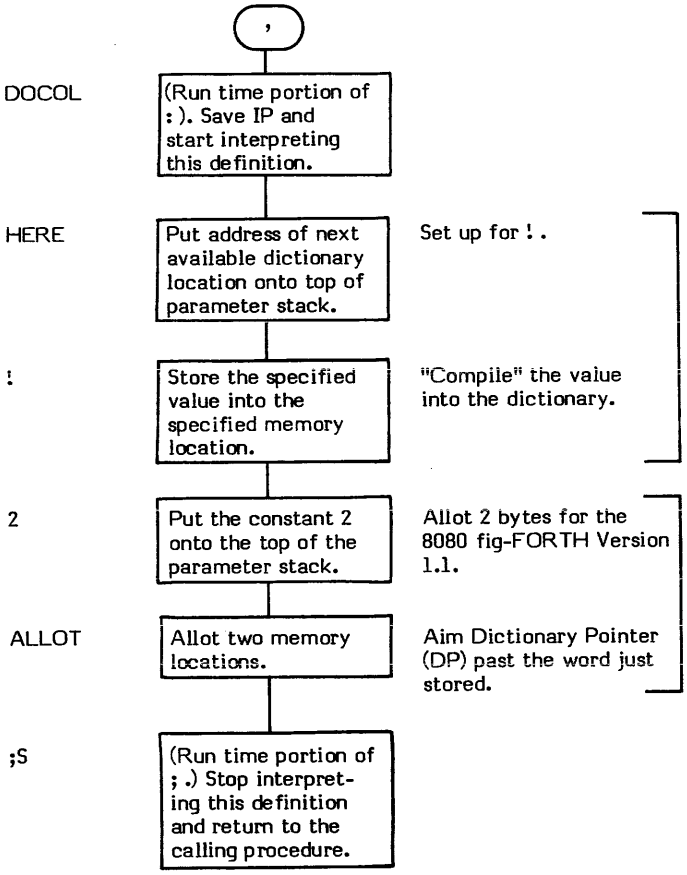
, is a high level colon definition.

Refer to INTERPRET .

~~FORTH-79~~: The FORTH-79 equivalent for , is , .

Definition:

```
: '  ( value -- )
     HERE !  2 ALLOT  ;
```



- (minuend\subtrahend -- difference)

- (pronounced "subtract" or "minus" in FORTH-79) is a signed single precision subtraction that subtracts the entry on the top of the stack from the second stack entry and replaces both values with their difference.

The high level subtraction is performed by two's complementing the subtrahend and then adding the two values together.

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value which is the subtrahend in the subtraction. The second stack entry contains a signed 16-bit single precision value which is the minuend in the subtraction.

* **At exit** - The top of the parameter stack contains the signed 16-bit single precision difference between the two input values.

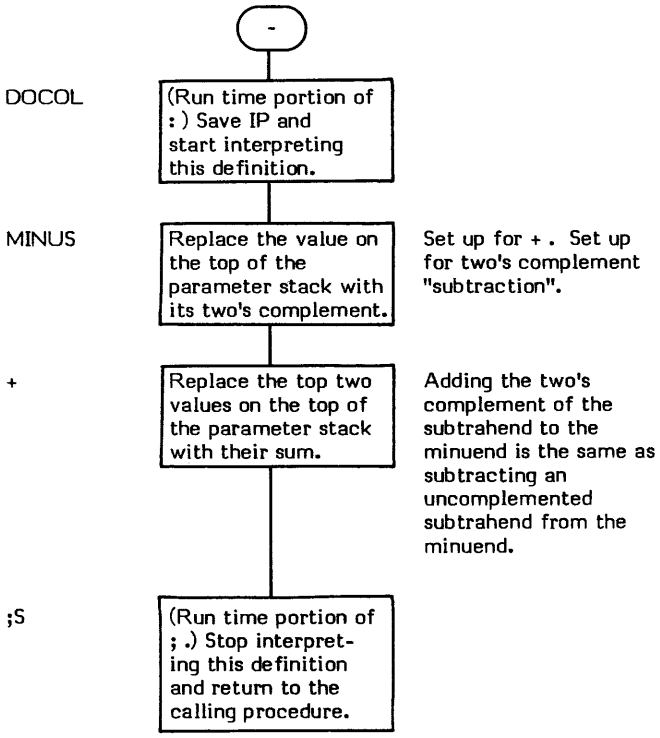
- is a high level colon definition.

Refer to MINUS .

FORTH-79: The FORTH-79 equivalent for - is - .

Definition:

```
:  -  ( minuend\ subtrahend -- difference )
     MINUS +  ;
```



--> (pronounced "next-block") is used at the end of a text screen in place of ;S to force interpretation to continue on to the next sequential screen.

Note: It is not "legal" to continue a colon definition onto the next screen (i.e., a ; must terminate a definition before a --> is issued).

As a matter of programming style, it is often desirable to create a "load screen" which specifically loads each screen via LOAD (with each LOAD followed by a comment describing the screen to be loaded) rather than loading multiple screens via --> . This makes it much easier to keep track of what is being loaded.

--> is an IMMEDIATE word since its purpose is to increment to the next screen during compilation. This means that its precedence bit is set and it will therefore execute at compile time.

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

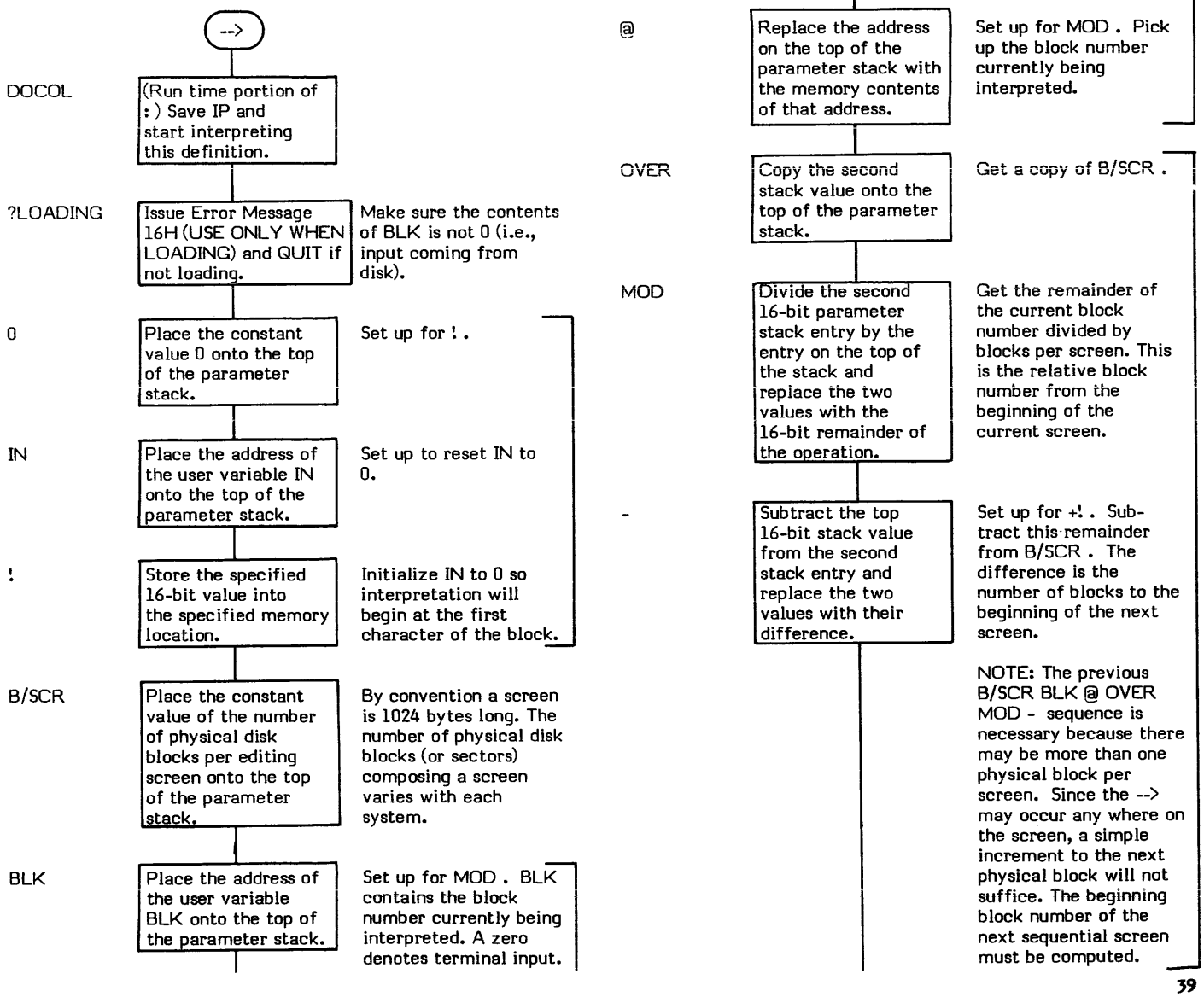
USE ONLY WHEN LOADING (16H) -- This word should only be used when loading.

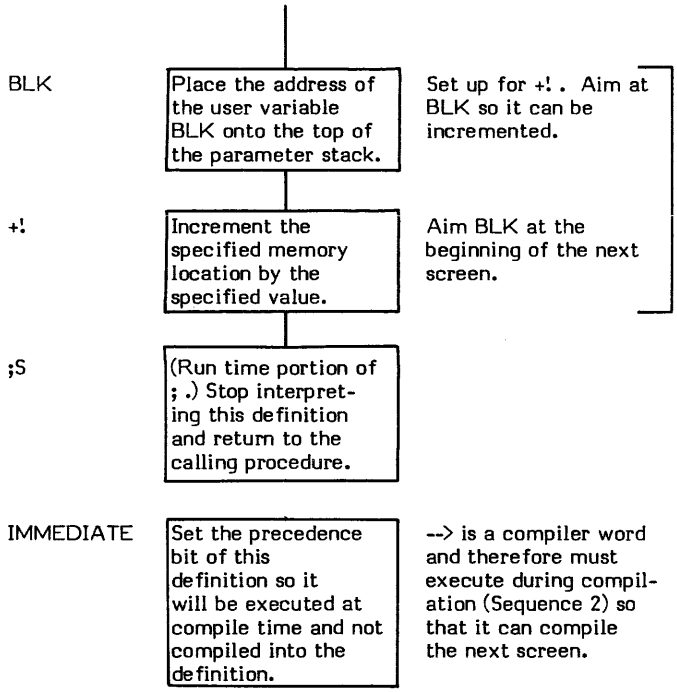
--> is a high level colon definition.

Refer to LOAD , and INTERPRET .

FORTH-79: --> is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition: : --> (--)
?LOADING 0 IN ! B/SCR BLK @ OVER MOD - BLK +! ;
IMMEDIATE




BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for +! . Aim at BLK so it can be incremented.
+!
Increment the specified memory location by the specified value.
Aim BLK at the beginning of the next screen.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of this definition so it will be executed at compile time and not compiled into the definition.
--> is a compiler word and therefore must execute during compilation (Sequence 2) so that it can compile the next screen.

-DUP

If zero: (value1 -- value1)

If non-zero: (value1 -- value1 \value1)

-DUP (pronounced "dash-dupe") duplicates the top of the parameter stack if its value is non-zero. For example:

0 -DUP will result in 0 being left on the stack.

102 -DUP will result in 102 102 on the stack.

* **At entry** - The top of the parameter stack contains a value to be duplicated.
* **At exit** - The value is duplicated if it is non-zero. In either case, the original value is on the top of the stack.

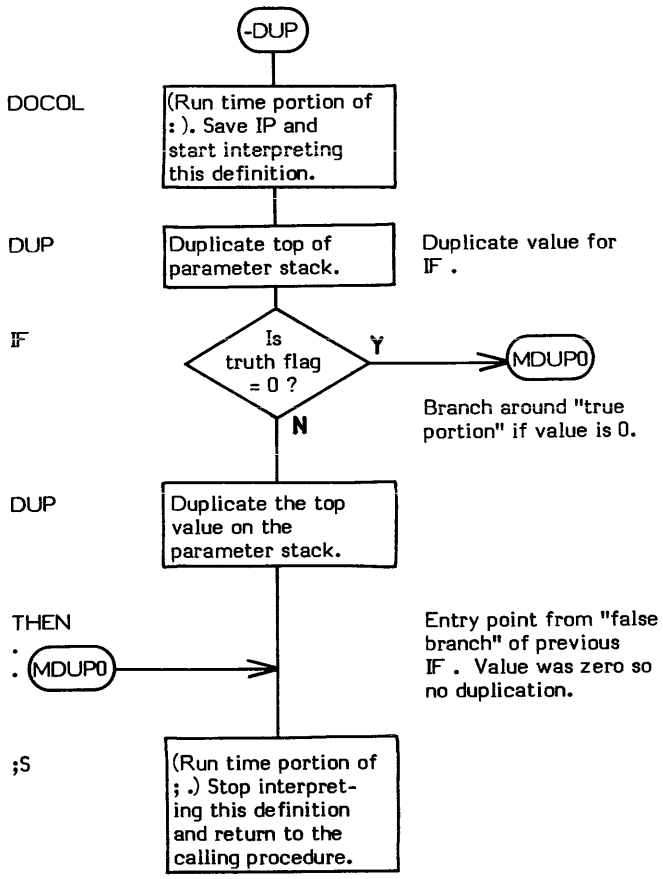
-DUP is a high level colon definition.

Refer to IF .

FORTH-79: The FORTH-79 equivalent for -DUP is ?DUP ("query-dupe").

Definition:

```
:  -DUP  ( value1 -- value 1  or  value 1\ value1 )
     DUP   IF   DUP   THEN   ;
```



-FIND

-FIND
Successful: (-- PFA\ length \ true flag)

Unsuccessful : (-- false flag)

-FIND (pronounced "dash-find") reads a word from the input stream and then searches the CONTEXT and CURRENT vocabularies for a definition whose name matches the input word.

The basis for -FIND is (FIND) .

* **At entry** - The next text word in the input stream (delimited by blanks) is used as the character string to search for. The parameter stack contains no input parameters.
* **At successful exit** - The top of the parameter stack contains a true flag (1). The second entry contains the matching definition's length byte. The third entry contains the matching definition's Parameter Field Address (PFA).
* **At unsuccessful exit** - The top of the parameter stack contains a false flag (0).

LIKELY ERROR MESSAGES:

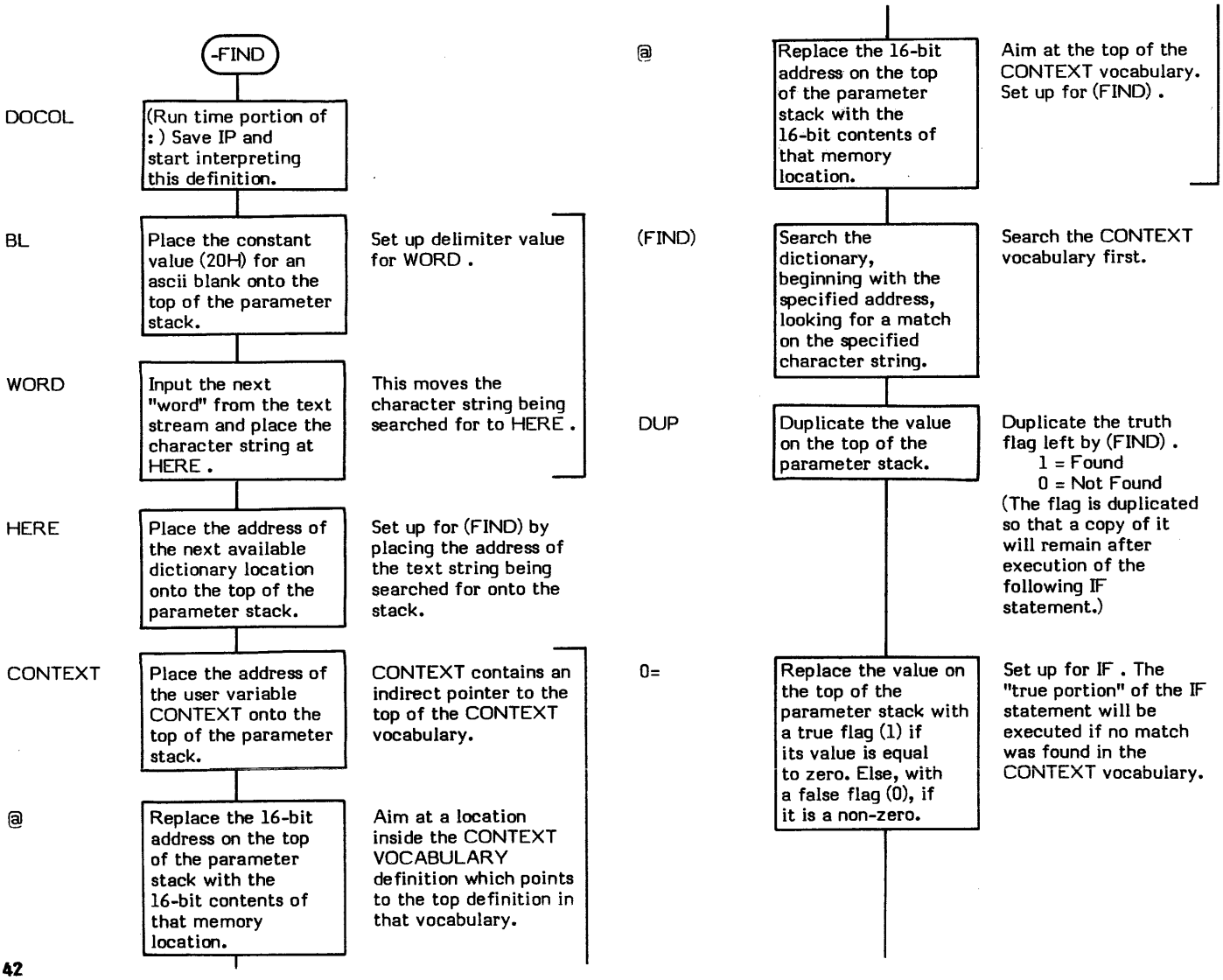
? pronounced "HUH?" (0) -- The word in question cannot be found in the dictionary.

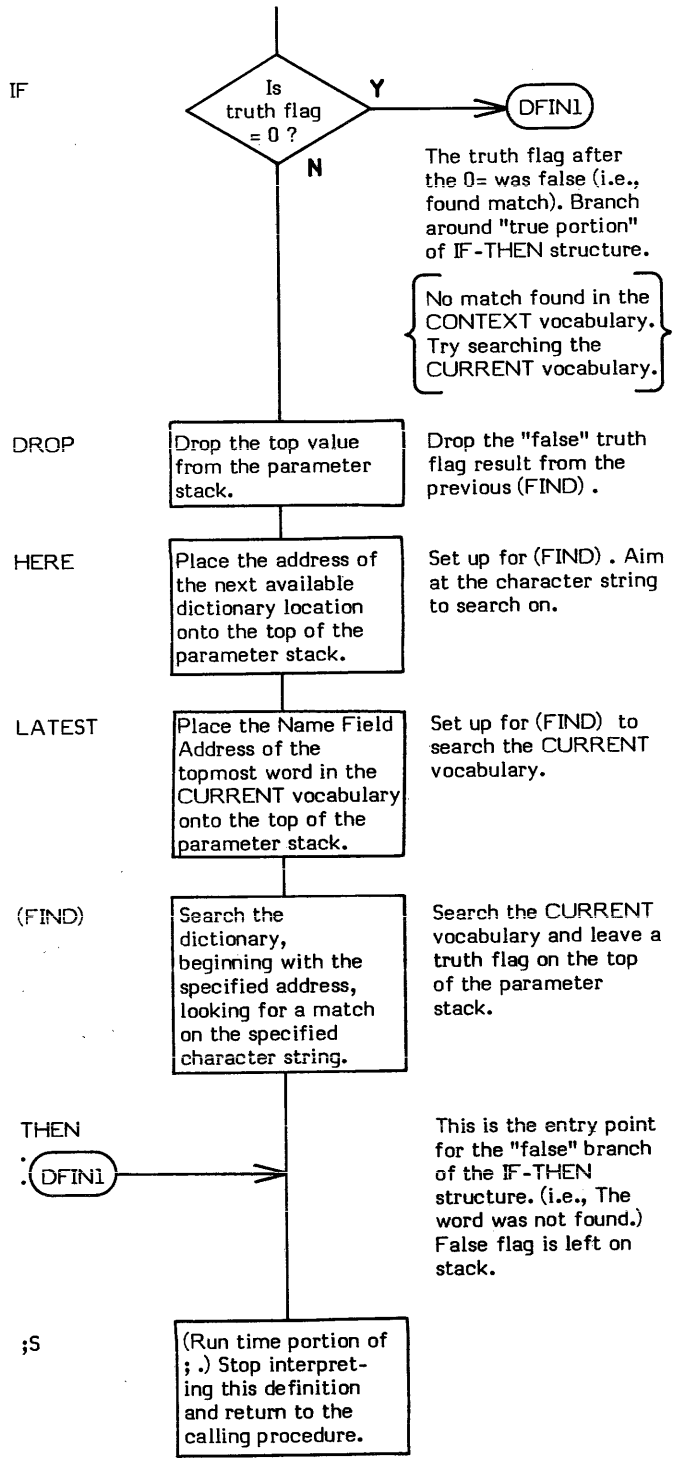
-FIND is a high level colon definition.

Refer to (FIND) , and WORD .

FORTH 79: The closest FORTH-79 equivalent to -FIND is FIND .

Definition: : -FIND (-- PFA\ length\ true flag)
(-- false flag)




IF
Is truth flag = 0 ?
Y
N
DFIN1
The truth flag after the 0= was false (i.e., found match). Branch around "true portion" of IF-THEN structure.
No match found in the CONTEXT vocabulary. Try searching the CURRENT vocabulary.
DROP
Drop the top value from the parameter stack.
Drop the "false" truth flag result from the previous (FIND) .
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
Set up for (FIND) . Aim at the character string to search on.
LATEST
Place the Name Field Address of the topmost word in the CURRENT vocabulary onto the top of the parameter stack.
Set up for (FIND) to search the CURRENT vocabulary.
(FIND)
Search the dictionary, beginning with the specified address, looking for a match on the specified character string.
Search the CURRENT vocabulary and leave a truth flag on the top of the parameter stack.
THEN
DFIN1
This is the entry point for the "false" branch of the IF-THEN structure. (i.e., The word was not found.) False flag is left on stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

-TRAILING

-TRAILING (beginning addr\ count -- beginning addr\ count)

-TRAILING (pronounced "dash-trailing") supresses trailing blanks when outputting a character string with TYPE . The input parameters for -TRAILING are identical to those for TYPE (as set up by COUNT). Using these parameters, -TRAILING examines the character string and reduces the character count so that trailing blanks are not output by a subsequent TYPE . -TRAILING scans backwards from the last character towards the first character. The first non-blank character encountered stops the scan.

.LINE is an example of a word which uses -TRAILING .

* **At entry** - The top of the parameter stack contains the character count of the text string including any trailing blanks. The second entry contains the true beginning address of the text string (i.e., the first character after the count byte).

* **At exit** - The top of the parameter stack contains the adjusted character count of the text string excluding any trailing blanks. The second entry contains the beginning address of the text string.

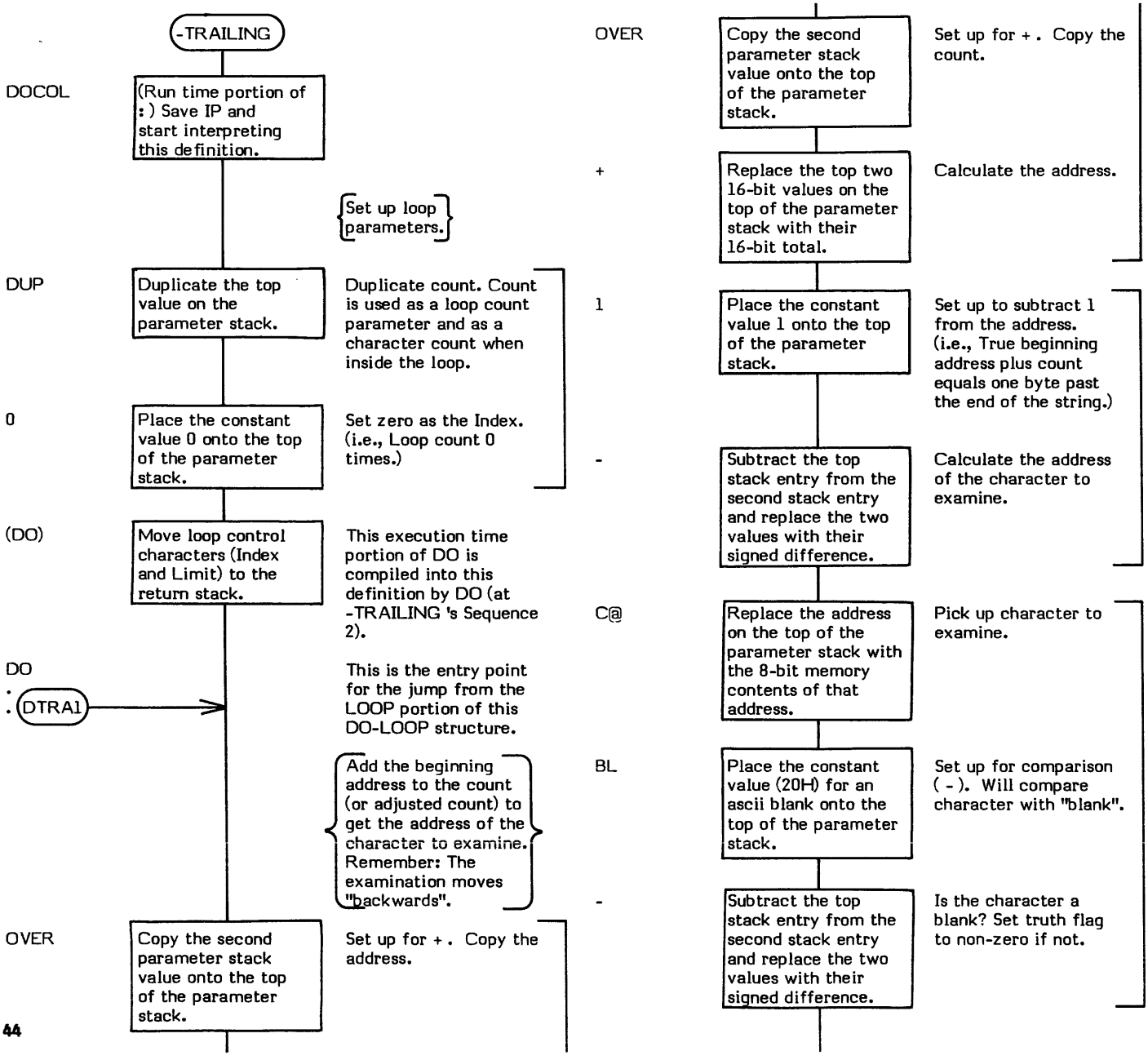
-TRAILING is a high level colon definition.

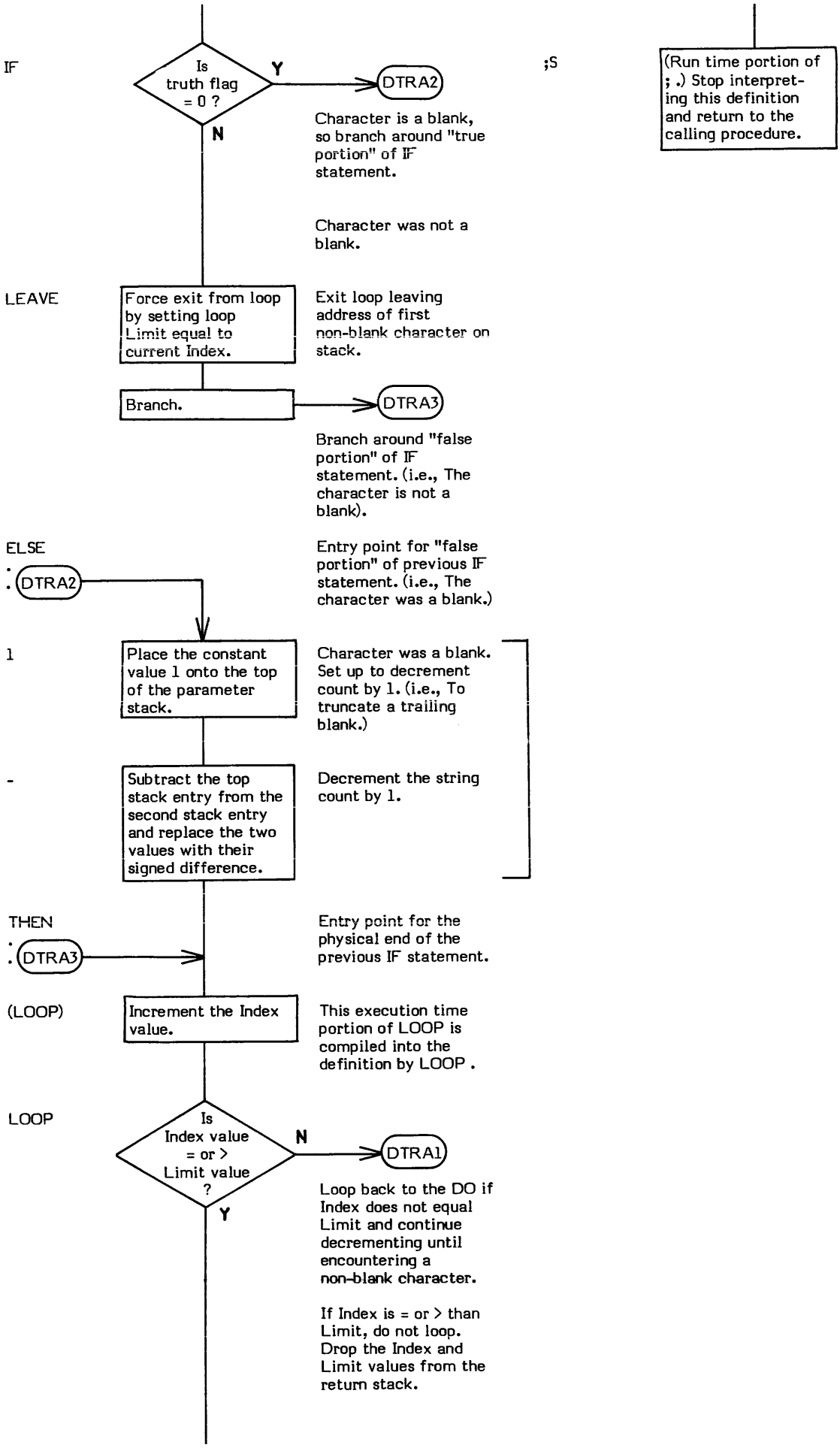
Refer to COUNT , and TYPE .

FORTH-79: The FORTH-79 equivalent for -TRAILING is -TRAILING .

Definition:

```
:   -TRAILING ( beginning addr\ count -- beginning addr\ count )
        DUP   0 DO
            OVER OVER +   1 -   C@ BL -
            IF   LEAVE
            ELSE   1 -
            THEN
        LOOP   ;
```




IF
Is truth flag = 0 ?
Y
N
DTRA2
Character is a blank, so branch around "true portion" of IF statement.
Character was not a blank.
LEAVE
Force exit from loop by setting loop Limit equal to current Index.
Exit loop leaving address of first non-blank character on stack.
Branch.
DTRA3
Branch around "false portion" of IF statement. (i.e., The character is not a blank).
ELSE
DTRA2
Entry point for "false portion" of previous IF statement. (i.e., The character was a blank.)
1
Place the constant value 1 onto the top of the parameter stack.
Character was a blank. Set up to decrement count by 1. (i.e., To truncate a trailing blank.)
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Decrement the string count by 1.
THEN
DTRA3
Entry point for the physical end of the previous IF statement.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
Y
DTRA1
Loop back to the DO if Index does not equal Limit and continue decrementing until encountering a non-blank character.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

■

. (value to be output --)

. (pronounced "dot") performs a binary-to-ascii conversion (pictured numeric output) on the 16-bit signed value on the top of the stack and prints the result (followed by one space) on the output device. The sign is printed only if the value is negative. The current value in BASE is used as the conversion radix.

NOTE: Since this is a signed conversion, problems arise when attempting to display 16-bit addresses which have their high order (sign) bit set. The set bit is interpreted as a negative bit and the value is displayed as a 15-bit negative number. This can be avoided by using U. which puts a zero onto the stack on top of the original value thereby making it a positive double precision value and then uses D. to print the result.

The basis of . is D. . (The pictured numeric words; <# , #S , SIGN , and #> ; are used to actually convert the value. These are located in D.R).

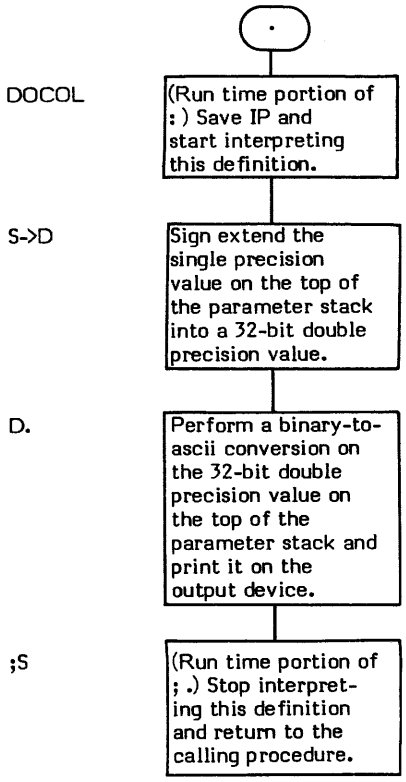
* **At entry** - The top of the parameter stack contains a signed 16-bit value to be converted and printed.
* **At exit** - No parameters.

. is a high level colon definition.

Refer to D. , and D.R .

FORTH-79: The FORTH-79 equivalent for . is . .

Definition: : . (value --)
S --> D D. ;



." (–)

." (pronounced "dot-quote") is a special FORTH word that exhibits both execution and compile time traits. The purpose of the word is to output a text string. When this string is output depends upon the state of the system when the word is executed. When execution begins, a test is made to determine if the system is in execution or compile state.

." is used in the form:

." text string "

where the terminating delimiter is a " (quote).

If the system is in execution state (i.e., ." is not inside of a definition), the text string is output immediately. If the system is in compile state, ." behaves as a standard compiler word and therefore exhibits two separate types of behavior: compile time and execution time behavior.

At compile time - the execution-time procedure address (that of (.")), the length of the string, and the string itself are compiled into the Parameter Field of the definition. (Refer to (.") for a description of what this definition looks like in the dictionary.)

At execution time - the procedure (.") outputs the text string.

The maximum allowable string length is installation dependent.

* **At entry** - ." must be followed by the character string to be output and delimited by " (quote).
* **At exit** - No parameters.

." is a high level colon definition.

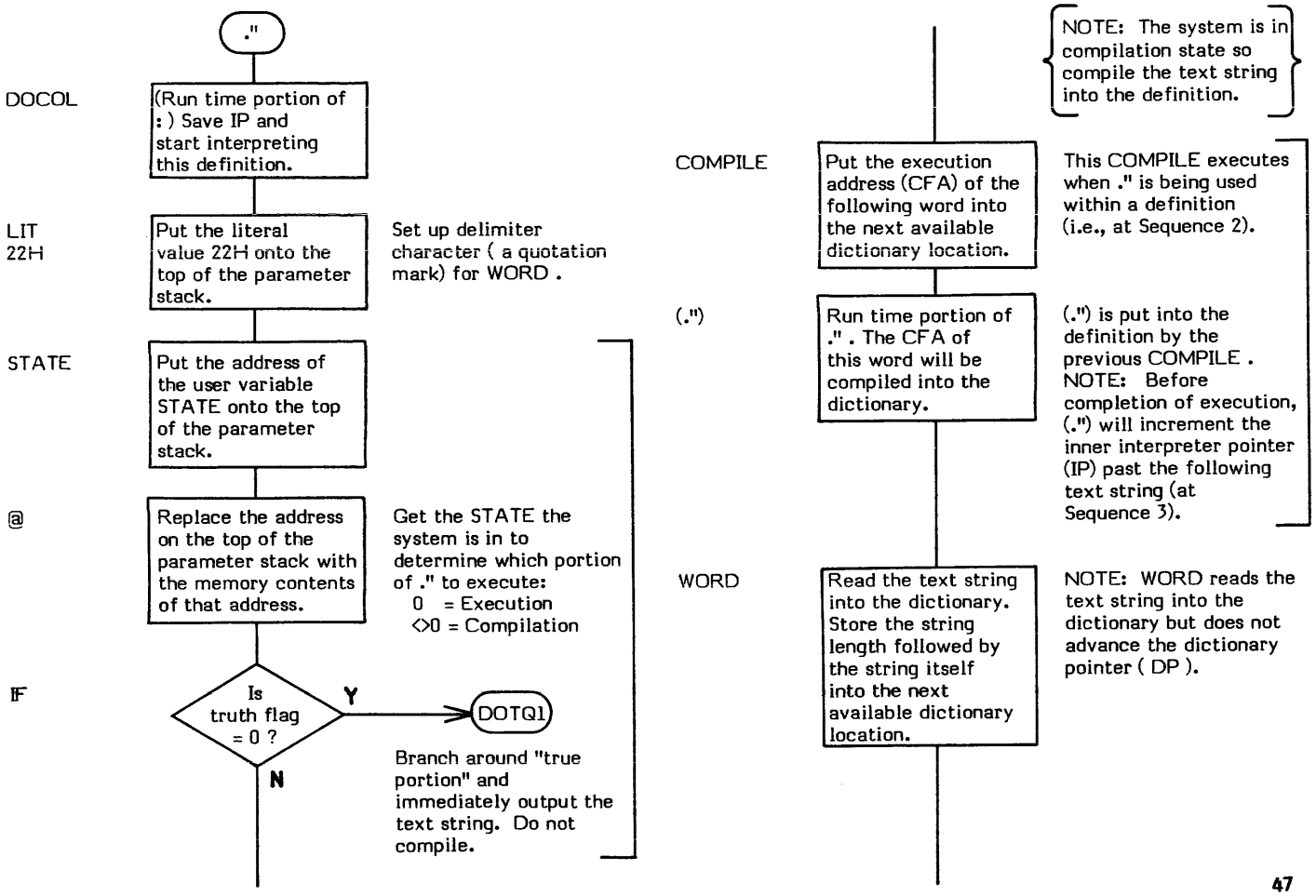
." is defined as IMMEDIATE and therefore has its precedence bit set causing it to execute during compilation.

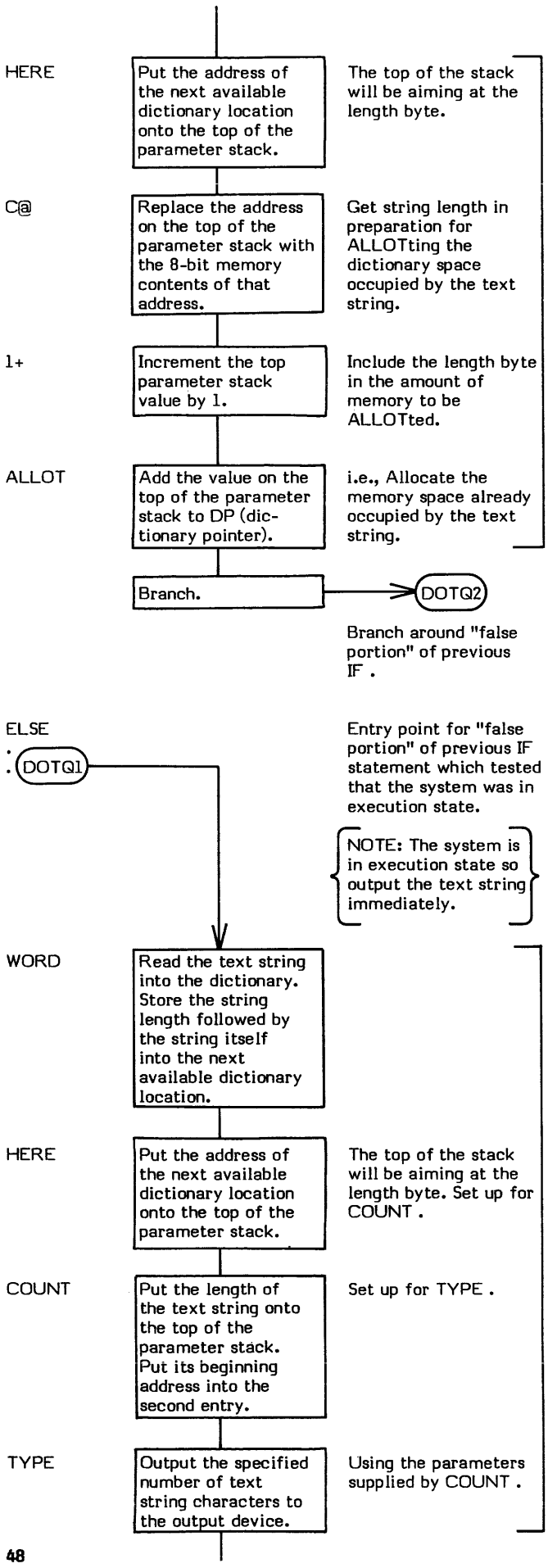
Refer to (.") .

FORTH-79: The FORTH-79 equivalent for ." is ." .

Definition:

```
:  ."  ( – )
     22  STATE @  IF  COMPILE (.")  WORD  HERE C@ 1+ ALLOT
                  ELSE  WORD HERE COUNT TYPE
                  THEN  ;  IMMEDIATE
```




HERE
Put the address of the next available dictionary location onto the top of the parameter stack.
The top of the stack will be aiming at the length byte.
C@
Replace the address on the top of the parameter stack with the 8-bit memory contents of that address.
Get string length in preparation for ALLOTting the dictionary space occupied by the text string.
1+
Increment the top parameter stack value by 1.
Include the length byte in the amount of memory to be ALLOTted.
ALLOT
Add the value on the top of the parameter stack to DP (dictionary pointer).
i.e., Allocate the memory space already occupied by the text string.
Branch.
DOTQ2
Branch around "false portion" of previous IF .
ELSE
. DOTQ1
Entry point for "false portion" of previous IF statement which tested that the system was in execution state.
NOTE: The system is in execution state so output the text string immediately.
WORD
Read the text string into the dictionary. Store the string length followed by the string itself into the next available dictionary location.
HERE
Put the address of the next available dictionary location onto the top of the parameter stack.
The top of the stack will be aiming at the length byte. Set up for COUNT .
COUNT
Put the length of the text string onto the top of the parameter stack. Put its beginning address into the second entry.
Set up for TYPE .
TYPE
Output the specified number of text string characters to the output device.
Using the parameters supplied by COUNT .
THEN
. DOTQ2
Entry point from branch around "false portion" of previous IF statement.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of this definition so it will be executed at compile time and not compiled into the definition.
." is a compiler word and therefore must execute during compilation (Sequence 2) so that it can compile other definitions.

.LINE (line # \ screen # —)

.LINE (pronounced "dot-line") prints a line of text from a screen stored on disk. Trailing blanks are suppressed and not printed.

The basis of .LINE is (LINE) . The constant C/L specifies the line length.

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value which is the desired screen number and the second entry contains a signed 16-bit single precision value which is the desired line number.
* **At exit** - No parameters.

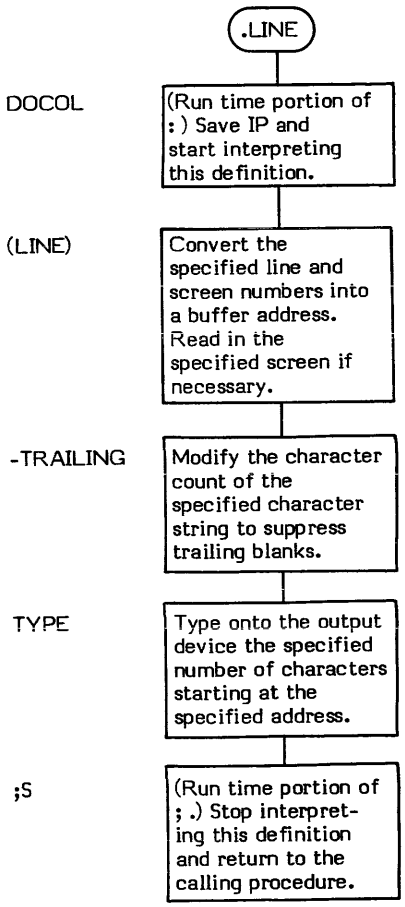
.LINE is a high level colon definition.

Refer to (LINE) .

FORTH-79: There is no FORTH-79 equivalent for .LINE .

Definition:

```
:  .LINE  ( line # \ screen # -- )
     (LINE)  -TRAILING  TYPE  ;
```



.R

.R (value \ field width —)

.R (pronounced "dot-R) performs a binary-to-ascii conversion (pictured numeric output) on the signed 16-bit value in the second stack entry and prints the result right justified in a field whose minimum width is specified by the value on the top of the stack. For example, if a field width of 10 is specified and only 3 characters are printed, the remainder of the field will be left padded with 7 blanks (b).

123 10 .R would result in bbbbbbb 123

Note, however, that the field width parameter specifies only a minimum field width. The entire value is always printed even if it exceeds the specified field width. No truncation occurs and no trailing blank is printed.

The current value in BASE is used as the conversion radix.

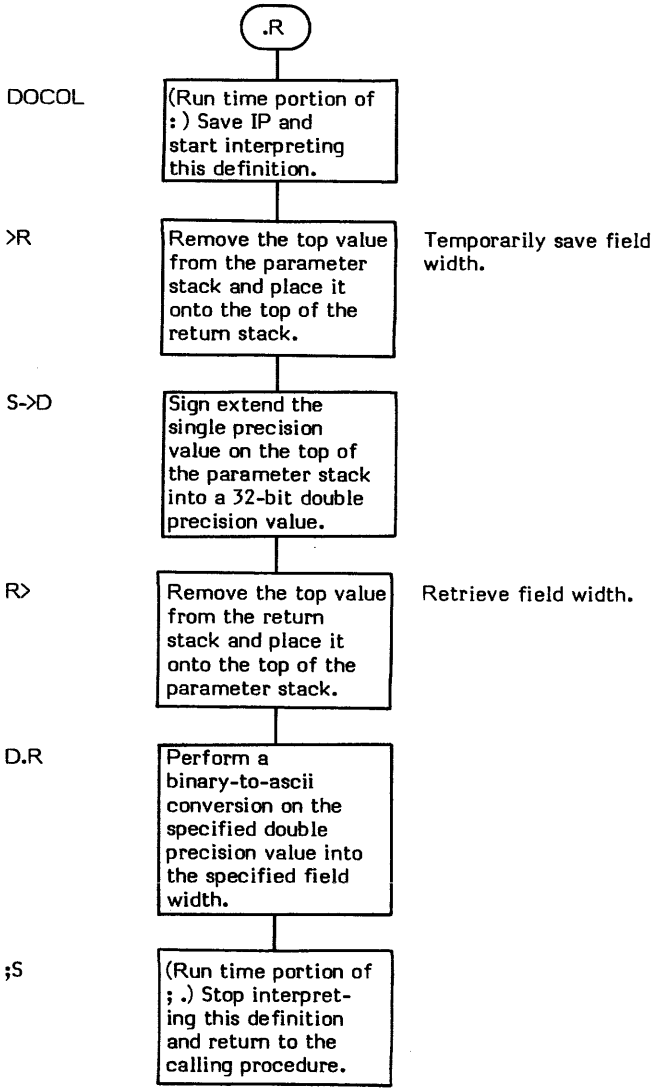
* **At entry** - The top of the parameter stack contains a signed 16-bit value which specified the field width of the converted ascii string. The second stack entry contains a signed 16-bit value to be converted and printed.
* **At exit** - No parameters.

.R is a high level colon definition.

Refer to D.R .

FORTH-79: .R is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:
```
:  .R  ( value\ field width )
       >R   S --> D   R>   D. R  ;
```



/ (dividend\ divisor — quotient)

/ (pronounced "divide") divides a 16-bit signed single precision value by another 16-bit signed single precision value and replaces them with their 16-bit signed quotient. The second stack value is divided by the top stack value.

The basis of / is /MOD . /MOD leaves a quotient and remainder. / drops this reminder. This is very similar to, but opposite from MOD , which drops the quotient.

* **At entry** - The top of the parameter stack contains a 16-bit signed single precision divisor. The second stack entry contains a 16-bit signed single precision dividend.

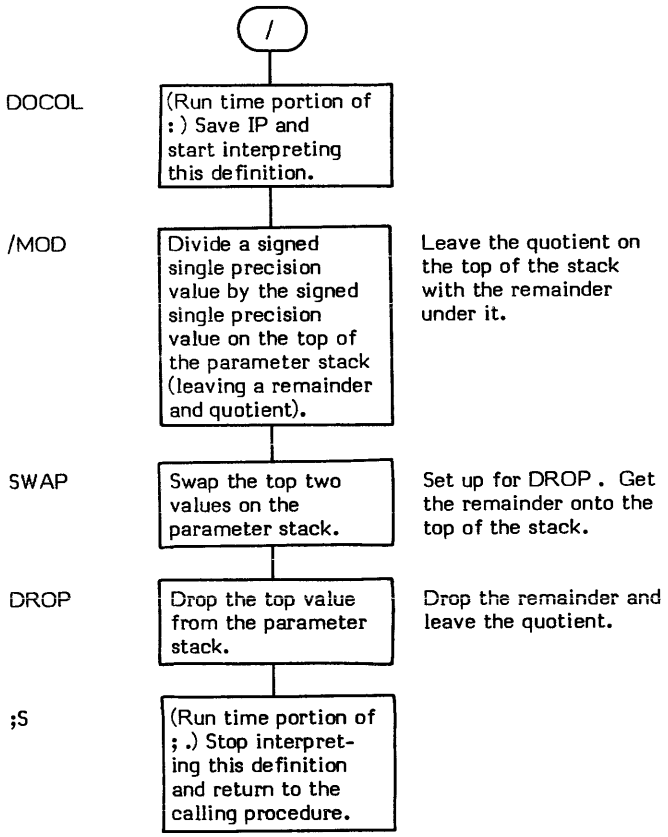
* **At exit** - The top of the parameter stack contains the 16-bit signed single precision quotient.

/ is a high level colon definition.

Refer to /MOD .

FORTH-79: The FORTH-79 equivalent for / is / .

Definition:
```
:  /  ( dividend\ divisor -- quotient )
      /MOD   SWAP   DROP   ;
```



/MOD

/MOD (dividend \ divisor -- remainder \ quotient)

/MOD (pronounced "divide-mod") divides a 16-bit signed single precision value by another 16-bit signed single precision value and replaces them with a 16-bit signed quotient and a 16-bit signed remainder. The remainder takes its sign from the dividend.

Note that /MOD uses a single precision dividend while M/ uses a double precision dividend.

The basis of /MOD is M/ . The single precision dividend is converted to double precision and then an M/ is preformed.

* **At entry** - The top of the parameter stack contains a 16-bit signed single precision divisior. The second stack entry contains a single precision 16-bit signed dividend.

* **At exit** - The top of the parameter stack containsthe 16-bit signed quotient. The second stack entry contains the 16-bit signed remainder.

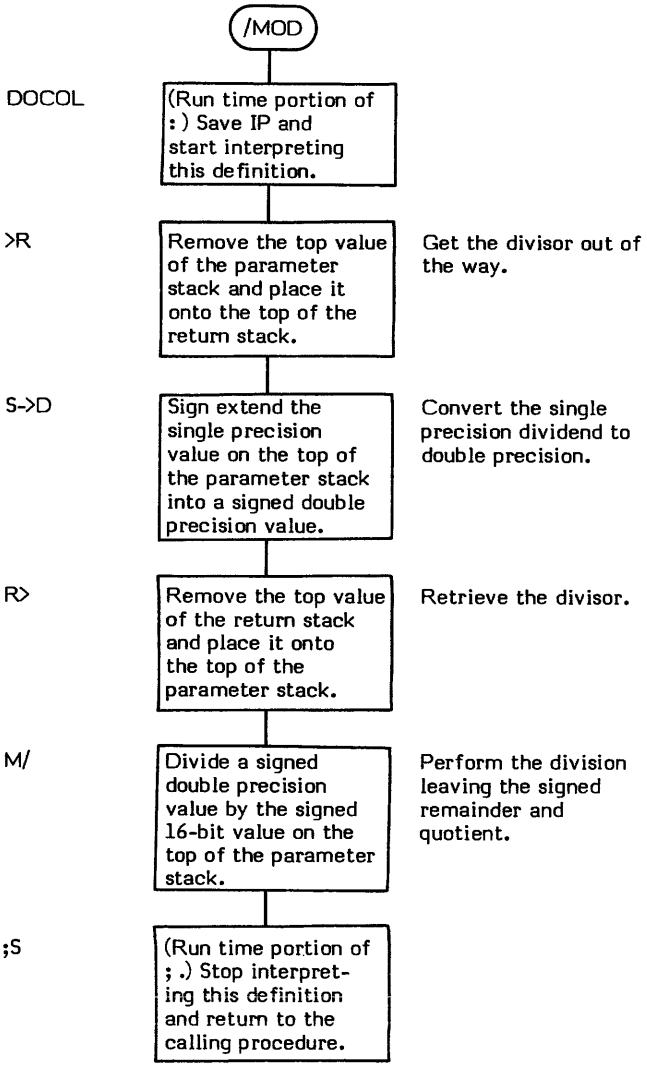
/MOD is a high level colon definition.

Refer to M/ .

FORTH-79: The FORTH-79 equivalent for M/ is M/ .

Definition:

```
:  /MOD  ( dividend \ divisor -- remainder \ quotient )
     >R   S --> D   R>   M/   ;
```



0 (— 0)

0 is a single precision CONSTANT value. This value is used so often that it has been made into a CONSTANT. This was done to save compiling time. Before converting a character string into a numeric value, INTERPRET first searches both the CONTEXT and CURRENT vocabularies. A significant amount of time can be saved if a name match can be found during the dictionary search instead of searching both vocabularies and then converting the value.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value 0.

FORTH-79: There is no FORTH-79 equivalent for 0 .

0<

0< (value — truth flag)

0< (pronounced "zero-less-than") examines a signed value on the top of the parameter stack and replaces it with a true flag (1) if the number is less than zero (negative) or with a false flag (0) if the number is greater than or equal to zero.

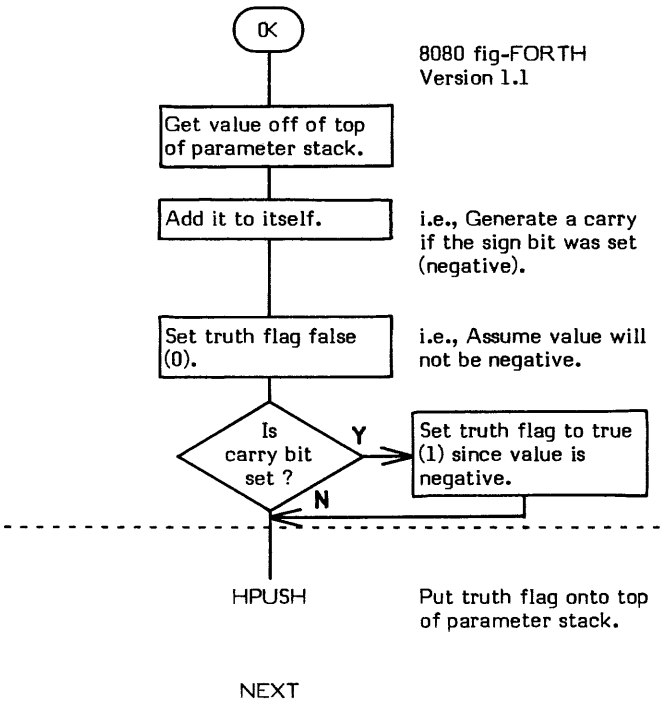
SIGN is an example of a word that uses 0< .

* **At entry** - The top of the parameter stack contains the signed 16-bit single precision value to be examined.
* **At exit** - The top of the parameter stack contains a truth flag.

0< is a low level code primitive.

Refer to SIGN .

FORTH-79: The FORTH-79 equivalent for 0< is 0< .



0= (value — truth flag)

0= (pronounced "zero-equal") examines the value at the top of the parameter stack and replaces it with a true flag (1) if the value is equal to 0 or with a false flag (0) if the value is not equal to 0.

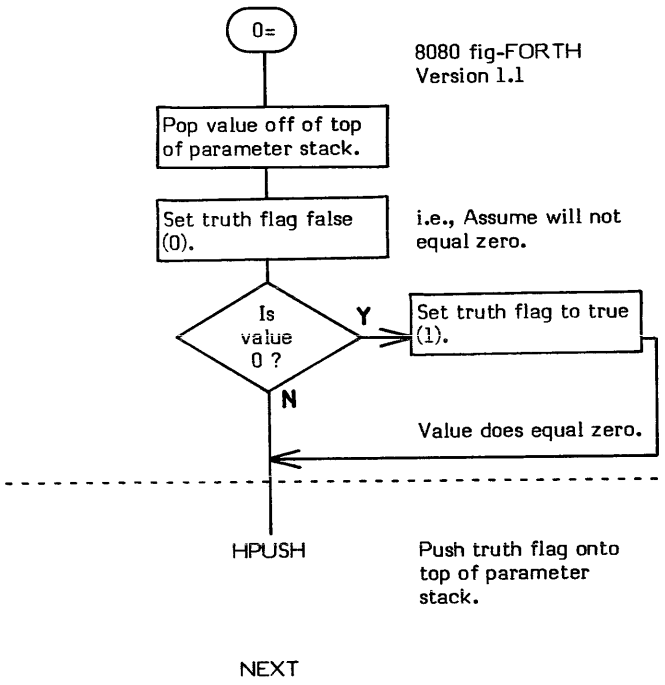
0= is often used to set up for an IF by complementing the truth flag so the statement's "true portion" (as opposed to the "false" or ELSE portion) can perform the desired function. This makes for more readable FORTH. In this instance the word NOT should be used for clarity. NOT is defined as:

```
: NOT  0= ;
```

* **At entry** - The top of the parameter stack contains the signed 16-bit single precision value to be examined.
* **At exit** - The top of the parameter stack contains a truth flag.

0= is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for 0= is 0= .



0BRANCH

0BRANCH (truth flag —)

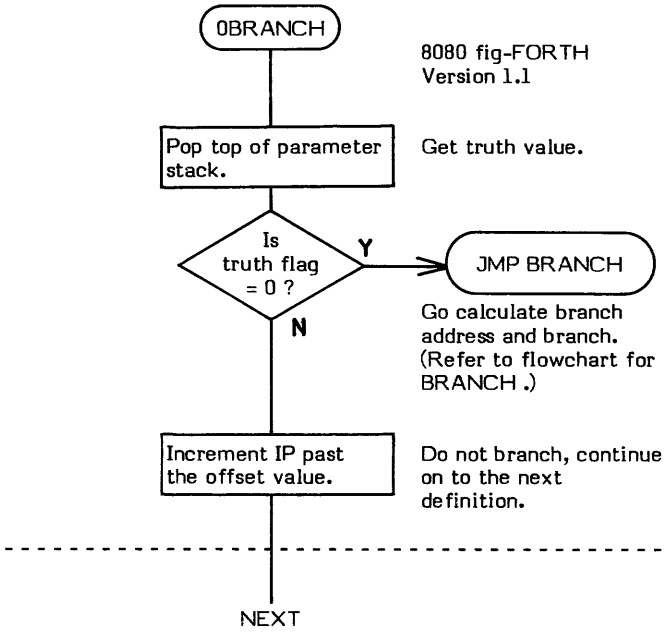
This execution time (Sequence 3) code (pronounced "zero-branch") is compiled into a definition (Sequence 2) by IF , UNTIL , and WHILE to perform a conditional branch. If the flag on the top of the parameter stack is 0 (false), the offset to which IP points at entry is added to IP to cause either a forward or backward branch depending upon the value of the offset (hence the name, 0BRANCH). If the flag is true, no branch is taken. Refer to BRANCH .

* **At entry** - The top of the parameter stack contains a 16-bit boolean truth flag.
* **At exit** - No parameters.

0BRANCH is a low level code primitive.

Refer to IF , UNTIL , WHILE , and BRANCH .

FORTH-79: There is no FORTH-79 equivalent for 0BRANCH .



1 (-- 1)

1 is a single precision CONSTANT value. This value is used so often that it has been made into a CONSTANT. This was done to save compiling time. Before converting a character string into a numeric value, INTERPRET first searches both the CONTEXT and CURRENT vocabularies. A significant amount of time can be saved if a name match can be found during the dictionary search instead of searching both vocabularies and then converting the value.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value 1.

FORTH-79: There is no FORTH-79 equivalent for 1 .

1+

1+ (value — value+1)

1+ (pronounced "one-plus") adds 1 to the value on the top of the parameter stack.

Note: The algebraic rules of signed addition apply here.

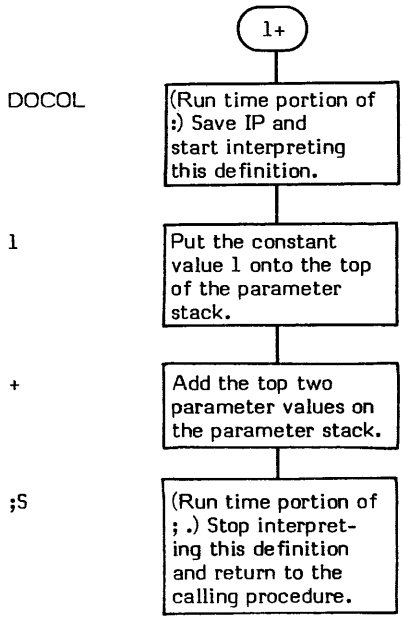
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value.
* **At exit** - The signed 16-bit single precision value on top of the parameter stack is incremented by 1.

1+ is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for 1+ is 1+ .

Definition:

```
: 1+ ( value -- value+1 )
   1  +  ;
```



2 (– 2)

2 is a single precision CONSTANT value. This value is used so often that it has been made into a CONSTANT. This was done to save compiling time. Before converting a character string into a numeric value, INTERPRET first searches both the CONTEXT and CURRENT vocabularies. A significant amount of time can be saved if a name match can be found during the dictionary search instead of searching both vocabularies and then converting the value.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value 2.

FORTH-79: There is no FORTH-79 equivalent for 2 .

2+

2+ (value — value+2)

2+ (pronounced "two-plus") adds 2 to the 16-bit value on the top of the parameter stack.

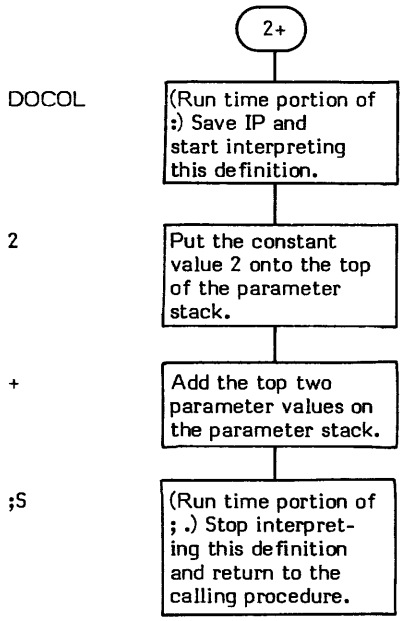
Note: The algebraic rules of signed addition apply here.

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value.
* **At exit** - The value on top of the parameter stack is incremented by 2.

2+ is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for 2+ is 2+ .

Definition:
```
:  2+  ( value -- value+2 )
     2   +   ;
```



3 (– 3)

3 is a single precision CONSTANT value. This value is used so often that it has been made into a CONSTANT. This was done tc save compiling time. Before converting a character string into a numeric value, INTERPRET first searches both the CONTEXT and CURRENT vocabularies. A significant amount of time can be saved if a name match can be found during the dictionary search instead of searching both vocabularies and then converting the value.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value 3.

FORTH-79: There is no FORTH-79 equivalent for 3 .

:

COMPILE TIME: (—)
(Sequence 2)

EXECUTION TIME: (—)
(Sequence 3)

: (pronounced "colon") is the defining word used to create (i.e., define) a high level FORTH definition. It is used in the following format:

: Specified name Body of definition ;

: is a defining word and therefore exhibits two different sets of action; those actions at compile time and those at run time.

The compile time (Sequence 2) action of : is:

1. To create a definition header in the dictionary for the name specified
2. To enter compile mode by setting the user variable STATE to a non-zero value (see STATE).
3. To set the CONTEXT vocabulary the same as the CURRENT vocabulary.

A colon definition is terminated with either ; or ;CODE .

The Code Address of the execution time portion of this word is compiled into the Code Field Address (CFA) of every colon definition word. 8080 Version 1.1 fig-FORTH references this routine via the label DOCOL . The purpose of this code is to save the current position of the Interpreter Pointer (IP) and begin execution of the body of this definition (i.e., thread "down" one level of threaded code).

The execution time (Sequence 3) action of DOCOL can be symbolically described in high level terms as follows:

IP @	Fetch the contents of IP . (IP points to the next CFA , i.e., "word" to execute.)
RP @ !	Fetch the address of the top of the return stack and store the address of the "next word to execute" on it.
-2 RP +!	Decrement the return stack pointer.
W @	Fetch the contents of W . (W points to the Code Field of the definition being executed. DOCOL is the Code Field procedure that is executing.)
2+ IP !	Increment W to point to the Parameter Field entry following the Code Field and store this address into IP. The system has chained down one level of nesting and the Parameter Field of this new definition will now be executed.

Note that : is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At Entry** - No parameters.
* **At Exit** - No parameters.

LIKELY ERROR MESSAGES:

? pronounced "HUH?" (0) -- The word in question cannot be found in the dictionary.

EXECUTION ONLY (12H) -- This word must not be used while the system is in compile mode.

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

: is itself a high level colon definition.

Refer to STATE , and VOCABULARY .

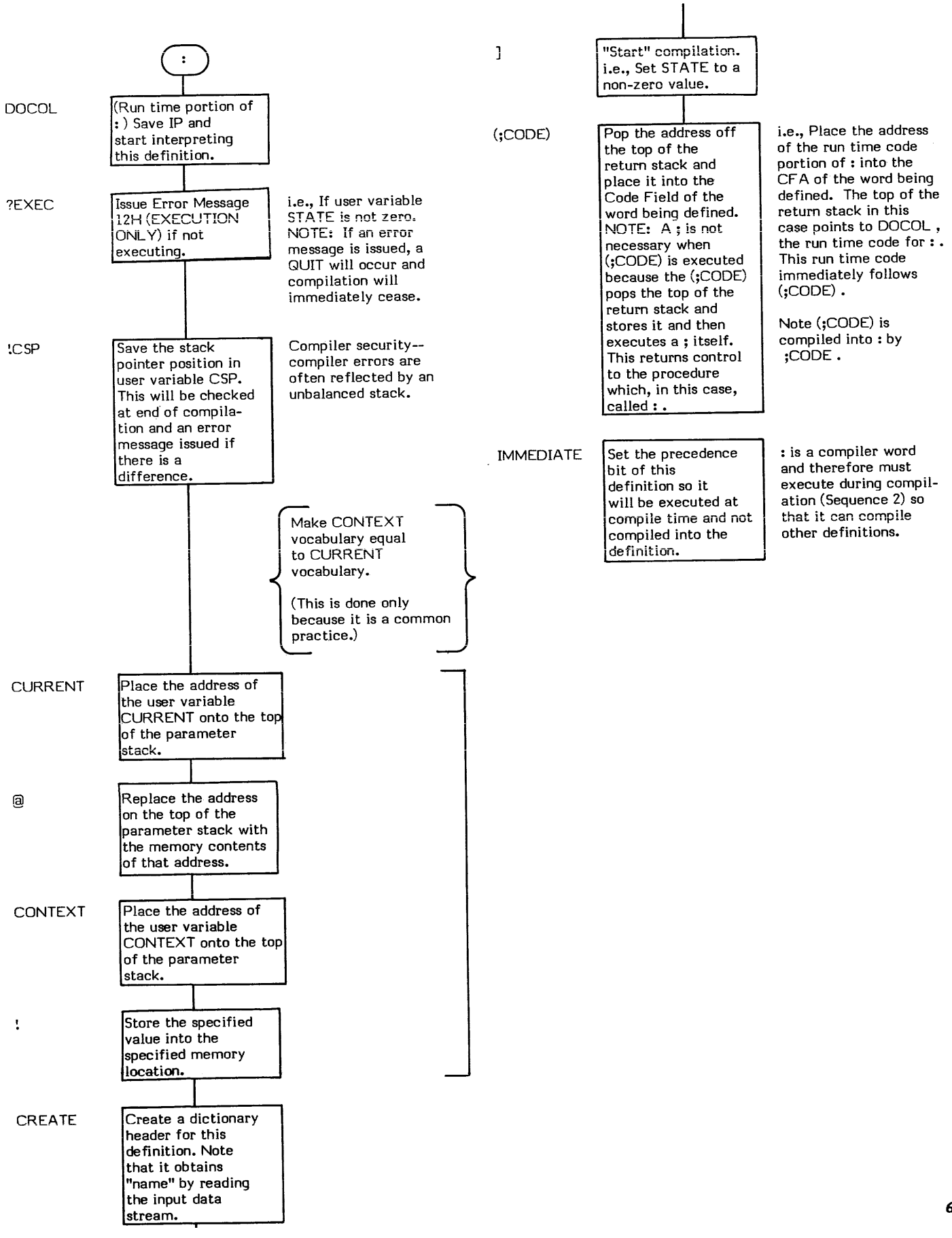
FORTH-79: The FORTH-79 equivalent for : is : .

Definition:

```
:  ( -- )  ( compile time )
     ?EXEC  !CSP  CURRENT @ CONTEXT !  CREATE  ]  (;CODE)

   Note: The (;CODE) is followed by assembly language sequence 3 time code.
     IMMEDIATE
```

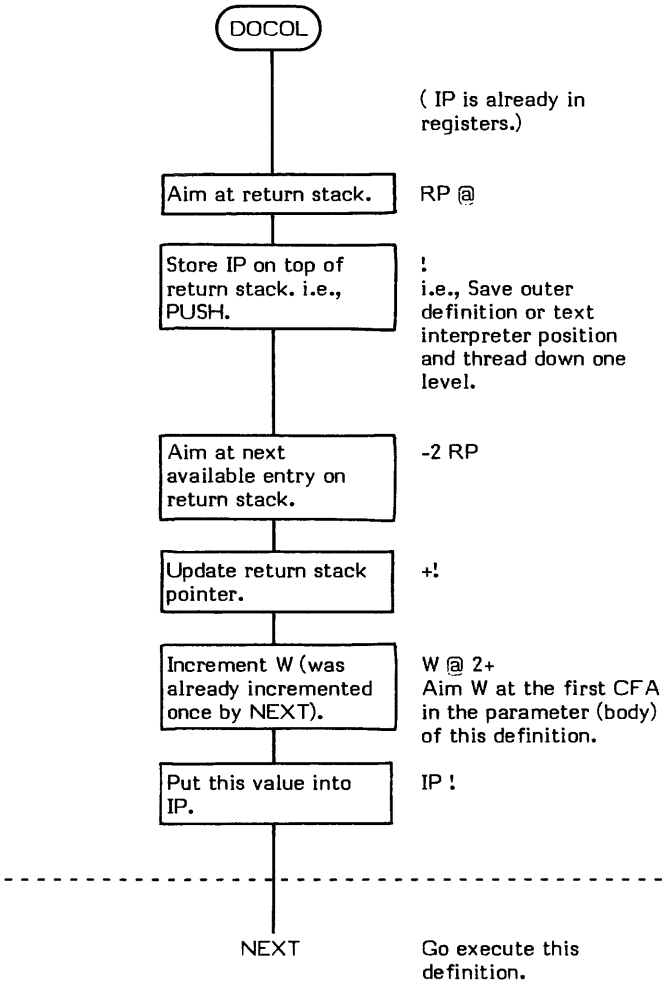
COMPILE TIME action of : (Sequence 2): (--)



EXECUTION TIME action of : (Sequence 3): (—)

NOTE: This code physically immediately follows the compile time (Sequence 2) code for : .

This is a high level representation of the 8080 assembly language code in the 8080 fig-FORTH Version 1.1 listing.



;

COMPILE TIME : (--)
(Sequence 2)

EXECUTION TIME: (--)
(Sequence 3)

; (pronounced "semicolon") is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

; is used to terminate a : ("colon") definition.

The compile time (Sequence 2) action of ; is:

1. To compile ;S (the execution time procedure of ;) into the definition being created.
2. To SMUDGE the smudge bit.
3. To place the system back into interpret mode (from compile mode) so that the next input data stream word can be interpreted.

The execution action (Sequence 3) of ; is actually that of ;S . Basically ;S "chains back" up one level of nesting and "returns control" to the definition that "called" the definition now executing ;S .

Note that ; is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** -No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

; is a high level colon definition.

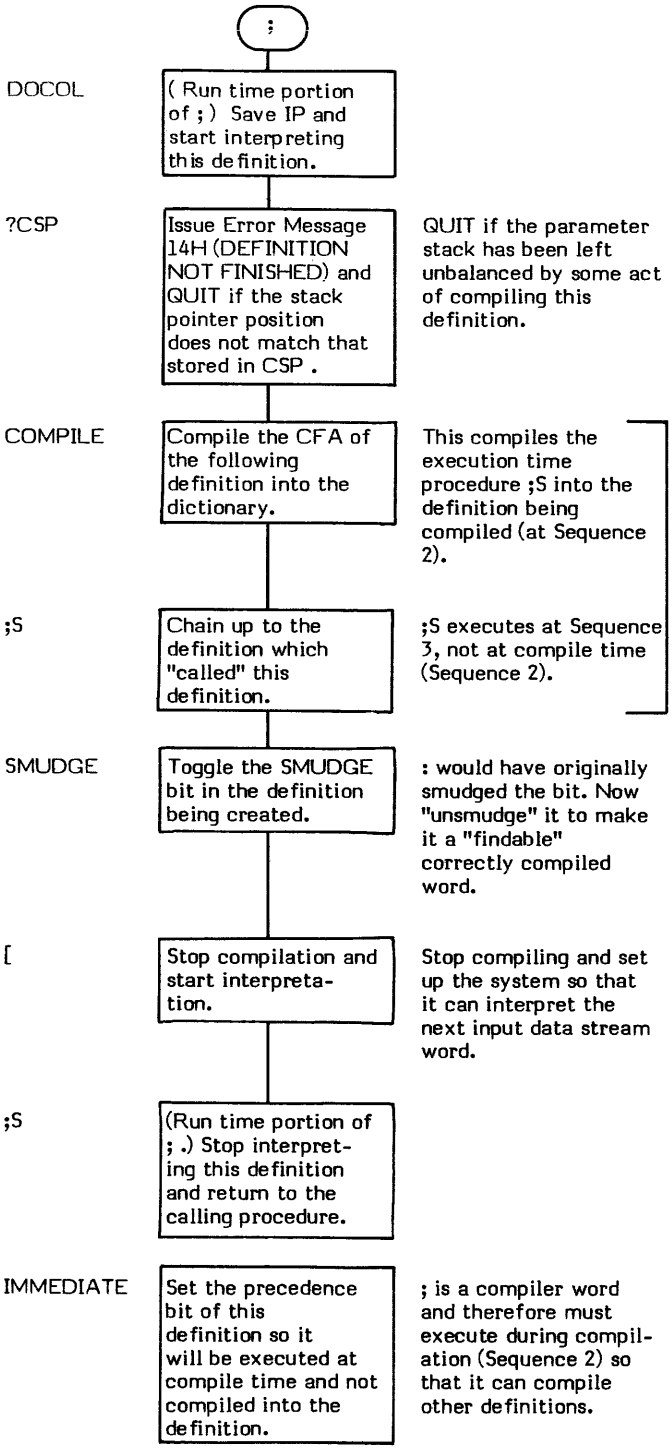
Refer to ;S , and : .

FORTH-79: The FORTH-79 equivalent for ; is ; .

Definition:

```
:  ;  ( -- )
   ?CSP   COMPILE ;S   SMUDGE   [   ;
```

COMPILE TIME action of ; (Sequence 2): (--)



EXECUTION TIME action of ; (Sequence 3): (--)

The execution time action of ; is that of ;S .

;CODE

DEFINITION TIME: (—)
(Sequence 1)

EXECUTION TIME: (—)
(Sequence 2)

;CODE (pronounced "semicolon-code") is a defining word and therefore exhibits two different sets of actions; those actions at definition time and those at execution time.

;CODE is a Sequence 1 defining word that is used to create Sequence 2 defining words. ;CODE is used in conjunction with another defining word (one that creates the header of the defining word being created) to create defining words ("parents") that in turn create other words ("children").

When creating a defining word (at Sequence 1), ;CODE acts as a dividing line between the high level words that create the "child" (at Sequence 2) and the assembly language code words that are the "child's" run time procedure (at the "child's" Sequence 3). See Figure ;CODE-1.

The function of ;CODE is very similiar to that of DOES> except ;CODE defines the beginning of a code procedure while DOES> defines the beginning of a high level procedure.

At Sequence 1 (i.e., when creating the "parent" defining word), ;CODE first compiles the address of its run time procedure, (;CODE) , (which actually executes at Sequence 2 when the "child" is created) and then stops compilation.

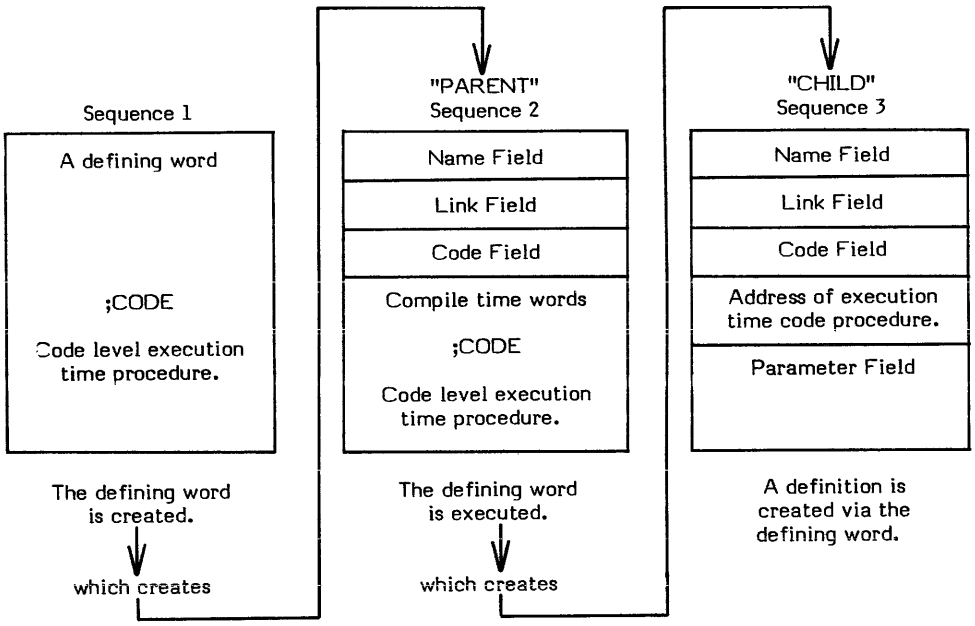


Figure ;CODE-1
Action of ;CODE

At Sequence 1 (i.e., when compiling the "parent" defining word), the high level words which will create the "parent" during Sequence 2 are compiled into the dictionary. ;CODE denotes the end of the "parent's" Sequence 2 compiling procedure and the beginning of the "child's" execution time procedure. Therefore, ;CODE must perform two main tasks:

1. It must first compile (;CODE) into the definition. This execution time procedure for ;CODE actually executes at Sequence 2 when the "parent" is creating the "child". Its function is to point the "child's" Code Field at the Sequence 3 code procedure that "lives" in the "parent" definition.

2. ;CODE must then stop compilation. This is done because prior to ;CODE are high level definitions whose addresses are compiled into the dictionary, but after ;CODE are assembler directives which must execute in order to assemble op-codes into the dictionary. These assembler directives are not IMMEDIATE . Therefore, the system must be taken out of compile state so the interpreter will execute these directives.

;CODE also SMUDGEs the header created by whatever defining word was used to begin the definition.

Note that the Version 1.1 8080 fig-FORTH does not set CONTEXT to the ASSEMBLER vocabulary. To rectify this problem, the word ASSEMBLER should immediately follow ;CODE .

;CODE only executes at Sequence 1. (;CODE) executes at Sequence 2. (Refer to (;CODE) .)

The end effect of ;CODE is that the dictionary definition structure of the "child" is determined by the words preceding ;CODE ; but, when the "child" executes at Sequence 3, the code following ;CODE is executed.

This allows whole new families of dictionary structures and their associated run-time codes to process these structures to exist. An example of a structure could be a graphics control table while the code following ;CODE could display the table.

Note that ;CODE is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

DEFINITION TIME (Sequence 1):

* **At entry** - No parameters.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

;CODE is a high level colon definition.

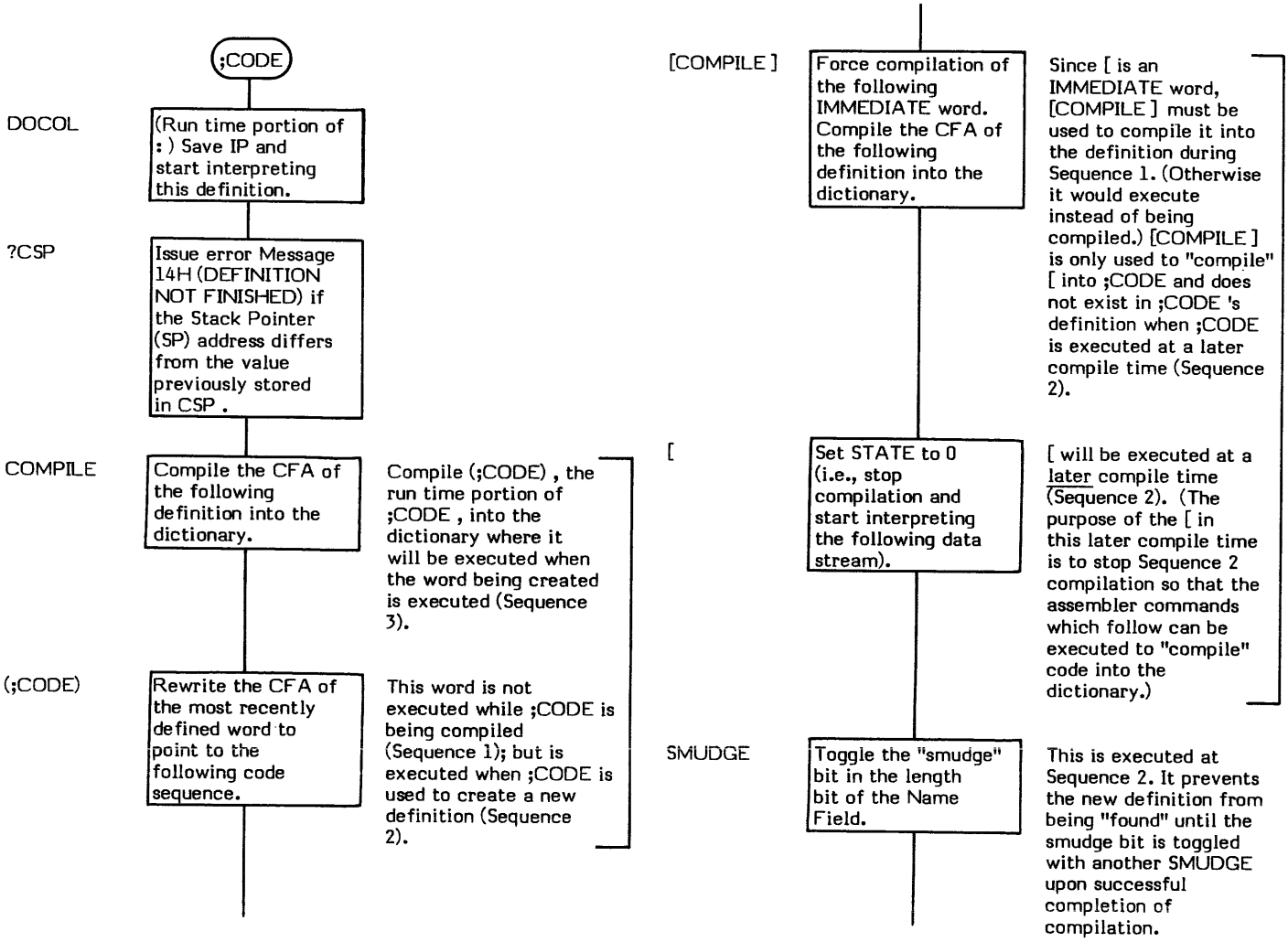
Refer to (;CODE) , DOES> , and <BUILDS .

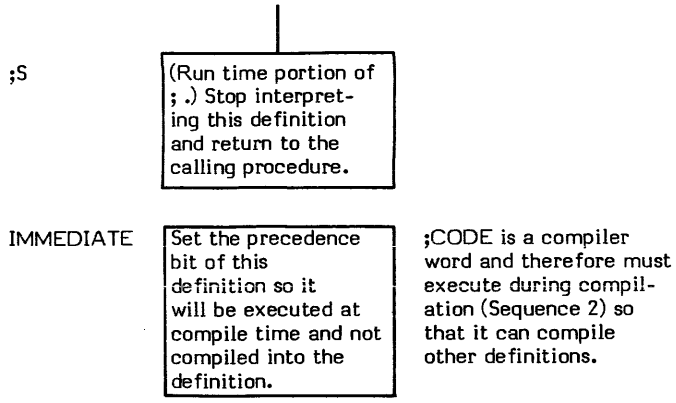
FORTH-79: There is no FORTH-79 equivalent for ;CODE .

Definition:

```
:  ;CODE  ( -- )
       ?CSP  COMPILE (;CODE)  [COMPILE ] [ SMUDGE  ;
       IMMEDIATE
```

DEFINITION TIME action of ;CODE (Sequence 1): (--)





EXECUTION TIME action of ;CODE (Sequence 2): (—)

The Sequence 2 action of ;CODE is to compile a (;CODE) into the definition being compiled.

The execution time (Sequence 3) action of ;CODE is (;CODE) .

;S

;S (–)

;S (pronounced "semicolon-S") is the execution time (Sequence 3) procedure compiled by ; . Its purpose, at the end of a colon definition, is to "chain back" to the next higher level by popping the return address of the calling procedure off of the return stack.

;S is also used to stop interpretation of a screen.

The action of ;S can be symbolically described in high level terms as follows:

RP @ Aim at the top of the return stack. (The top of the return stack contains the "return" address. i.e., The address of the next "word" to execute located within the Parameter Field of the definition which "called" the definition this ;S is in.)

@ IP ! Fetch the address of the next "word" to execute and store it into IP .

2 RP +! Increment the return stack pointer (i.e., free up this location).

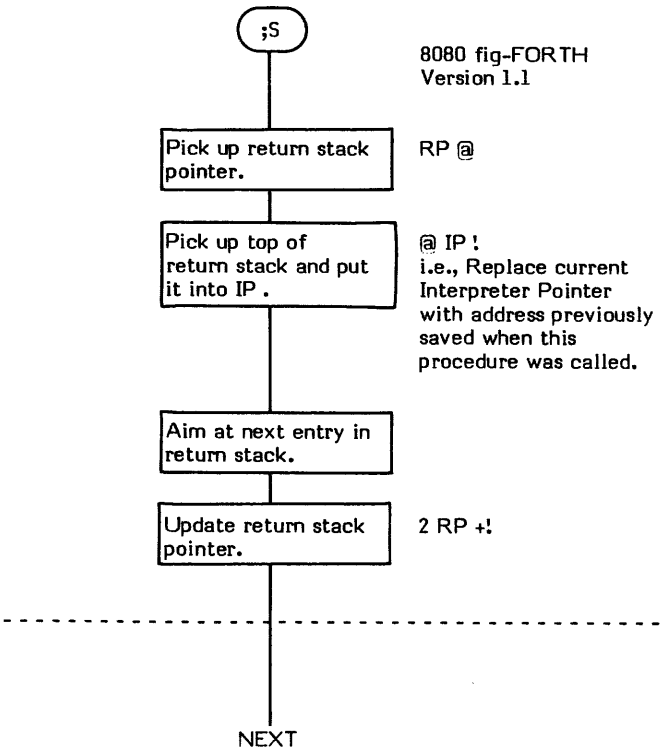
IP is all set to execute the next "word". The last thing ;S does is execute NEXT (which then executes the "next" word).

* **At entry** - No parameters.
* **At exit** - No parameters.

;S is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for ;S is EXIT .

Note: This is a high level representation of the 8080 assembly language code in the 8080 fig-FORTH Version 1.1 listing.



< (value1 \ value2 -- flag)

< (pronounced "less-than") performs a signed comparison of the top two single precision values on the parameter stack and replaces them with a boolean truth flag. The flag is true (non-zero) if the second stack entry is less than the top stack entry (hence the name "less-than"). Otherwise the flag is false (0) if the second stack entry is equal to or greater than the top entry. Note that this is just like an in-fix operation in which the operator has been moved outside. For example:

5 < 4 (in-fix) is the same as 5 4 < (post-fix)

MAX is an example of a word which uses < .

This is a signed comparison.

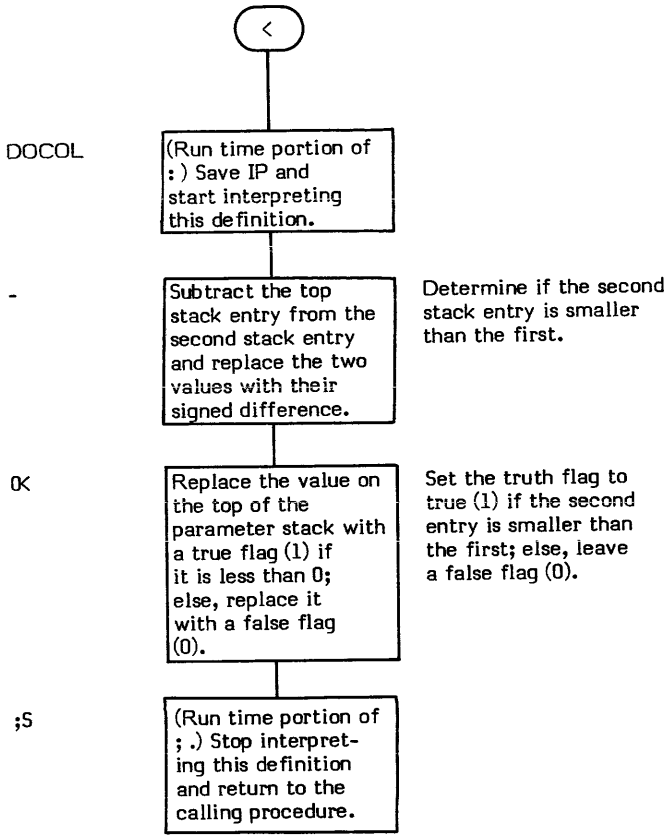
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value to be compared with the second entry which is also a signed 16-bit single precision value.
* **At exit** - The top of the parameter stack contains a true boolean flag (non-zero) if the second stack entry is less than the top of the stack or a false boolean flag (0) if it is greater.

< is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for < is < .

Definition:

```
: < ( value1 \ value2 -- flag )
    - 0< ;
```



<#

<# (–) (Refer to "At entry" section)

<# (pronounced "less-than-sharp") begins a double precision integer pictured numeric conversion expression.

Pictured numeric conversion converts values on the stack into ascii strings which are formatted according to picture specifications. This is similiar in concept to the BASIC PRINT USING or COBOL PICTURE statements.

The character string is created in memory, starting one byte before PAD and working backwards toward low memory. The conversion takes place one digit at a time starting with the one's column, then the ten's column, etc. (That is why memory is filled in a right-to-left direction.)

The user variable HLD contains a pointer to the last converted character (see HOLD).

For example, the general structure of a pictured numeric expression is:

<#	#	#S	SIGN	#>
start	one digit	multiple digits	sign	end

Some examples are:

<# # # # #S #> gives at least 4 digits.

<# # # 2EH HOLD #S #> gives 2-decimal places.

The phrase <# #S SIGN S> is used by the word D.R .

The description of # describes the binary-to-ascii conversion process in more detail.

The specific purpose of <# is to initialize the pointer in HLD to aim at the beginning of PAD . (<# can be modified if it is desirable to create the string elsewhere; e.g., in the memory of a memory mapped CRT display, etc.). Once executed, <# should not be used again until a #> has been executed as this would cause converted characters to overlay previously converted characters in PAD . Also, no words should be executed during pictured conversion which would cause the end of the dictionary, hence PAD , to change location.

NOTE: Although this word by itself requires no parameters, correct FORTH coding techniques suggest that parameters not be introduced during the string conversion process. Therefore, pictured numeric conversion parameters should be set up prior to the execution of this word.

* **At entry** - Although this word does not require any entry parameters, the top of the parameter stack should contain a 31-bit value to be converted and output (see #) with its high order portion in the first entry and the low order portion in the second entry. (Note the 31-bit value. # correctly converts only positive values. Therefore a DABS should normally precede the <# to convert negative double precision values to their absolute values.) If the output character string is to be signed, an optional 16-bit signed value may be located in the third stack entry. This parameter sets up for SIGN . (Refer to SIGN for a complete description of this parameter and how to set it up.)

* **At exit** - This word by itself leaves no output parameters on the stack. Refer to #> for output of the pictured numeric conversion sequence.

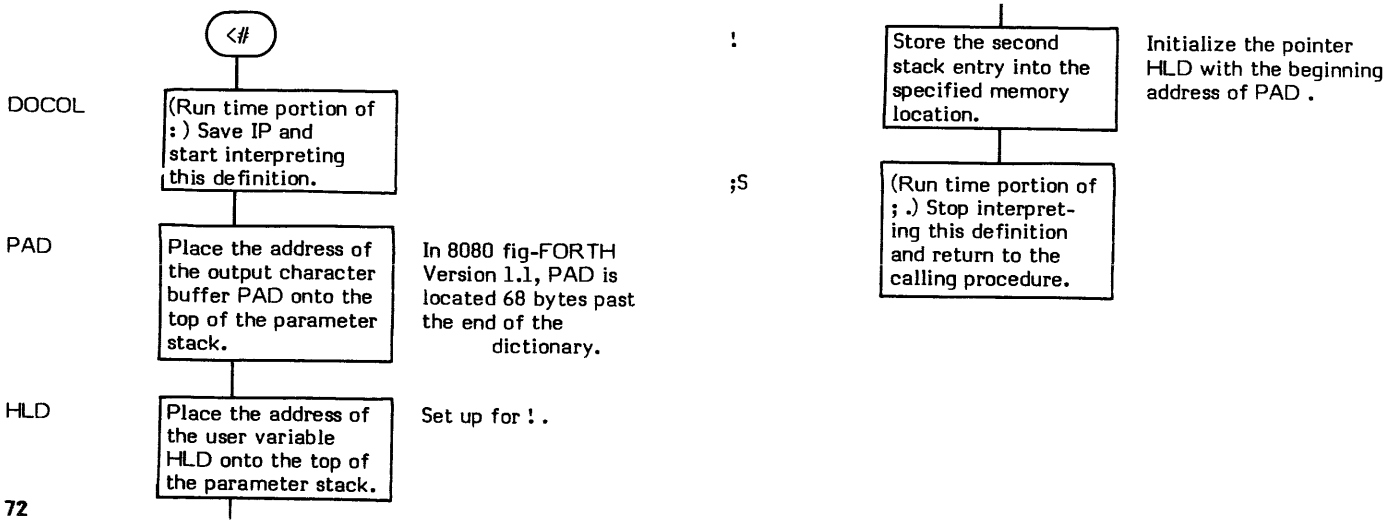
<# is a high level colon definition.

Refer to # , #> , SIGN , HLD , and HOLD .

FORTH-79: The FORTH-79 equivalent for <# is <# .

Definition:

```
:  <#  ( -- )
       PAD  HLD  !  ;
```



<BUILDS

DEFINITION TIME: (—)
(Sequence 1)

EXECUTION TIME: (—)
(Sequence 3)

<BUILDS (pronounced "builds") is a defining word and therefore exhibits two different sets of actions; those actions at definition time and those at execution time.

<BUILDS is a Sequence 1 defining word which is used to create Sequence 2 defining words. Words created by <BUILDS are then used to create other words. e.g., <BUILDS (Sequence 1) is used to create the defining word VOCABULARY (Sequence 2) which in turn creates vocabulary definitions such as FORTH , EDITOR , and ASSEMBLER . Such a word is then executed (Sequence 3) to set the point at which dictionary searches start.

<BUILDS is normally used in conjunction with DOES> . When used this way, <BUILDS "builds" (hence the name) a dictionary header for the new definition. Refer to Figure <BUILDS-1. This header is actually the definition for the constant 0. Definitions between <BUILDS and DOES> are executed at compile time.

The words between <BUILDS and DOES> are executed at Sequence 2 time while creating a new definition. The words following DOES> are executed at Sequence 3 time when executing the new definition. (Refer to Figure <BUILDS-1.)

The compile time (Sequence 2) action of <BUILDS is to simply "build" a dictionary header for the new definition. This header is actually the definition for the constant 0. DOES> then overlays this header with execution time (Sequence 3) pointers for the word (refer to DOES>). The Code Field Address is overlayed to point to the execution time code for DOES> . The first entry in the Parameter Field (the 0) is overlayed with the address of the new definition's execution time (Sequence 3) definitions. This is the address of the first word following DOES> .

The action of <BUILDS and DOES> can be more easily understood by observing how the words VOCABULARY and FORTH are created and used. VOCABULARY is created at Sequence 1 with <BUILDS and DOES> . The words following DOES> are compiled into the dictionary at Sequence 1 but will not be executed until Sequence 3 as explained later. Then, at Sequence 2, when VOCABULARY is executed to create a new definition (named FORTH in this example); the following happens:

1. <BUILDS creates a constant 0 header, with the first Parameter Field entry filled in with the 0 value. (This is important to note because this 0 will be overlayed by DOES> with the address of the execution time definitions for FORTH , which in turn means that the first Parameter Field entry in a <BUILDS-DOES> definition is not available for use.)

2. The words following <BUILDS are also executed at Sequence 2 to build (compile) the vocablulary definition.

3. The Pseudo Name Field is compiled into the definition.

4. The Vocabulary Link Field is compiled into the definition.

5. The Chronological Link Field is compiled into the definition. (<BUILDS has no Sequence 3 time action. i.e., When FORTH is executed.)

6. After the Chronological Link Field is compiled, DOES> executes. Remember this is the compile time (Sequence 2) time action of DOES> . DOES> first overlays the Code Field of the definition being created, FORTH , with the Code Field Address of DOES> 's execution time (Sequence 3) code.

The 0 in the first Parameter Field entry is overlayed with the address of the next dictionary location following DOES> . This is the address of the execution time procedure (Sequence 3) of the example word, FORTH . The words following DOES> were compiled at Sequence 1 into the VOCABULARY definition.

Note this distinction: The execution time (Sequence 3) procedure for the example word, FORTH, (the definition created by VOCABULARY at Sequence 2); exists in VOCABULARY , not in FORTH . The first entry Parameter Field in FORTH points to this execution time procedure.

When FORTH is executed at Sequence 3, its Code Field points to the execution time (Sequence 3) code for DOES> . The execution time purpose of DOES> is to transfer control to the execution time procedure (remember DOES> is for high level definitions) for the example, FORTH . It does this by fetching FORTH 's execution time address from the first Parameter Field entry and placing it onto the return stack and calling NEXT . This code also causes the address of the second Parameter Field entry to be placed onto the top of the parameter stack.

The execution time procedure for FORTH is then executed physically in the last half of VOCABULARY , then control returns to the word following FORTH .

DEFINITION TIME (Sequence 1):

* **At entry** - No parameters.

* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) -- The dictionary has grown into the Terminal Input Buffer.

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

<BUILDS is a high level colon definition.

Refer to DOES> , VOCABULARY , FORTH , and ;CODE .

FORTH-79: The FORTH-79 equivalent for <BUILDS is CREATE . Refer to FORTH-79 Standard.

Definition:

```
: <BUILDS  ( -- )
    0 CONSTANT  ;
```

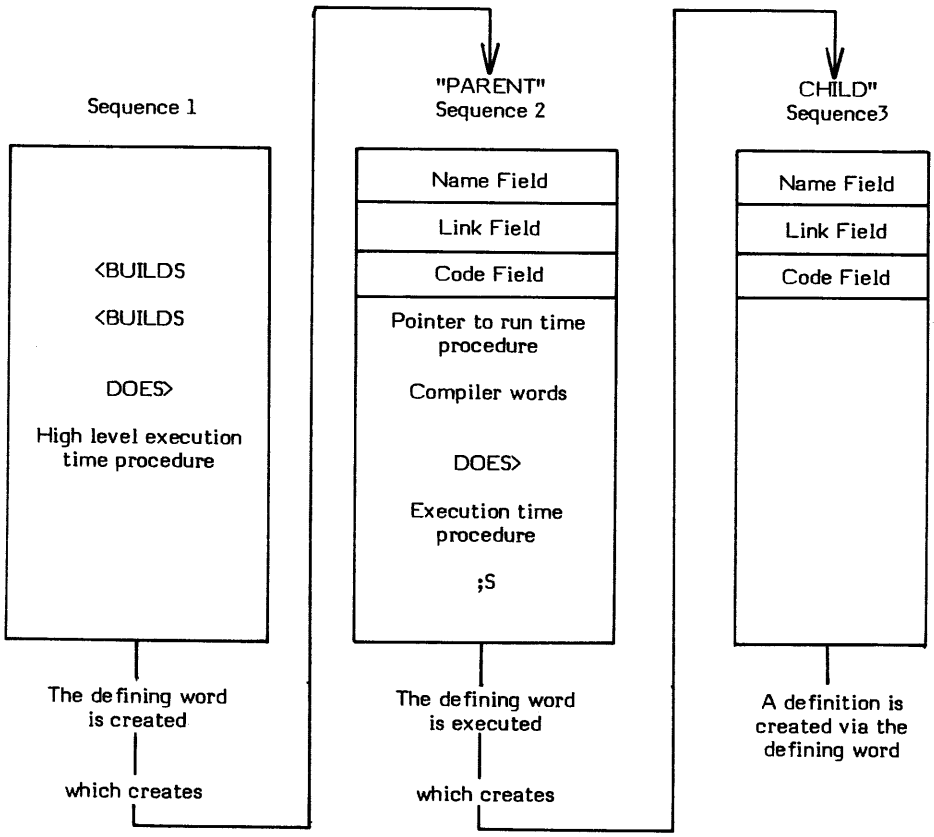
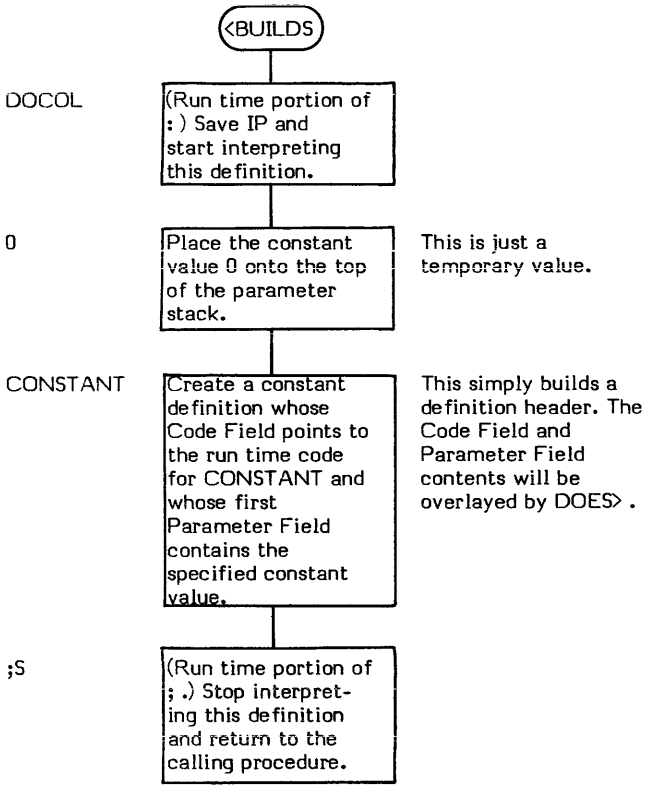


Figure <BUILDS-1
Compile Time Action of <BUILDS and DOES>

DEFINITION TIME action of <BUILDS (Sequence 1): (—)



EXECUTION TIME action of <BUILDS: (—)

<BUILDS simply creates a header for a definition and therefore has no Sequence 2 or Sequence 3 time action.

=

= **(value1 \ value2 -- flag)**

= (pronounced "equals") compares the top two values on the parameter stack and replaces them with a boolean flag. The flag is true (1) if the values are equal. The flag is false (0) if the values are not equal.

EXPECT is an example of a word which uses = .

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value to be compared for equality with the second stack entry, which is also a signed 16-bit single precision value.

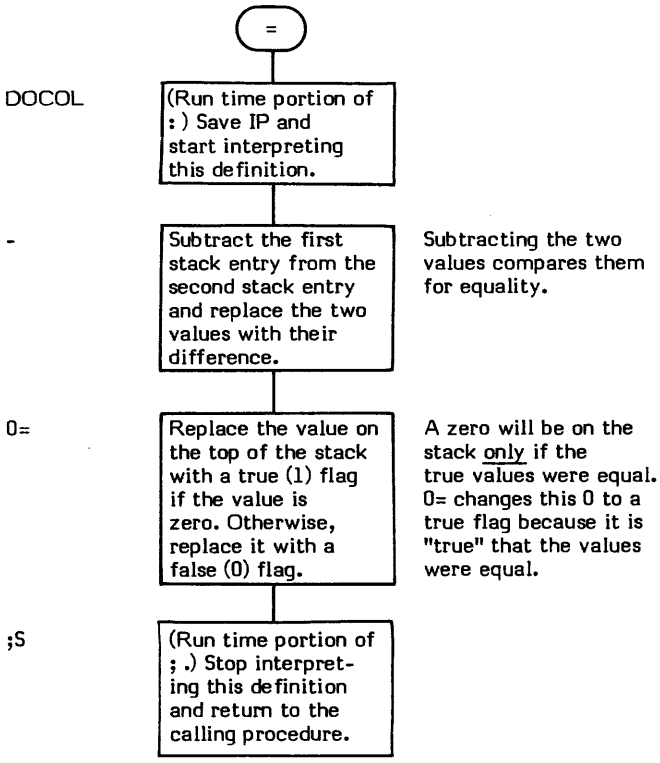
* **At exit** - The top of the parameter stack contains a boolean flag. The flag is true (1) if the input values were equal and false (0) if they were not equal.

= is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for = is = .

Definition:

```
:  =  ( value1 \ value2 -- flag )
     - 0=  ;
```



> (value1\ value2 -- truth flag)

> (pronounced "greater-than") compares the two top-most signed 16-bit values on the parameter stack and replaces them with a truth flag. The truth flag is set true (1) if the second value is greater than the top of the stack value and false (0) if it is less than or equal to the top of the stack value.

At execution time, > simply swaps parameters and performs a < comparison.

* **At entry** - The top and second parameter stack entries contain the signed 16-bit single precision values to be compared.
* **At exit** - The top of the parameter stack contains a truth flag. This flag is set true (1) if the second stack value was greater than the top of the stack value. Otherwise, it is set false (0).

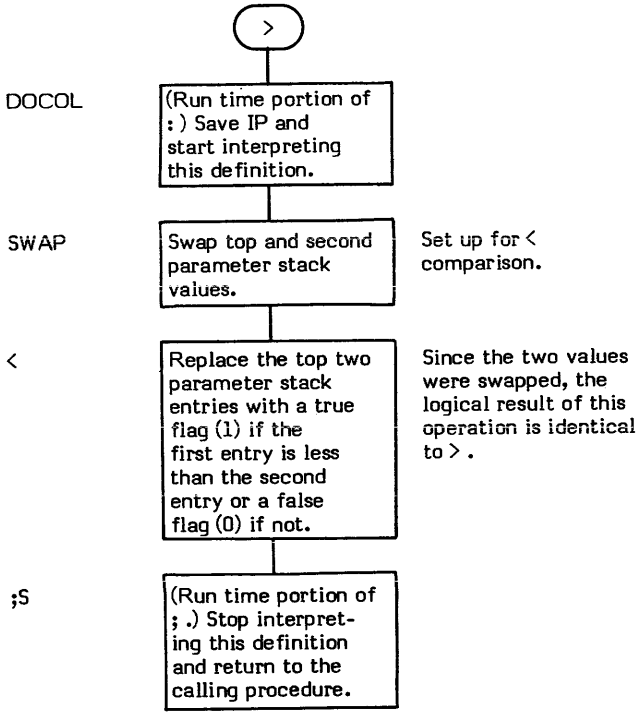
> is a high level colon definition.

Refer to < .

FORTH-79: The FORTH-79 equivalent for > is > .

Definition:

```
: >  ( value1 \value2 -- flag )
     SWAP <  ;
```



>R

>R (value to be placed onto return stack —)

>R (pronounced "to-R") pops a number from the top of the parameter stack and pushes it onto the top of the return stack.

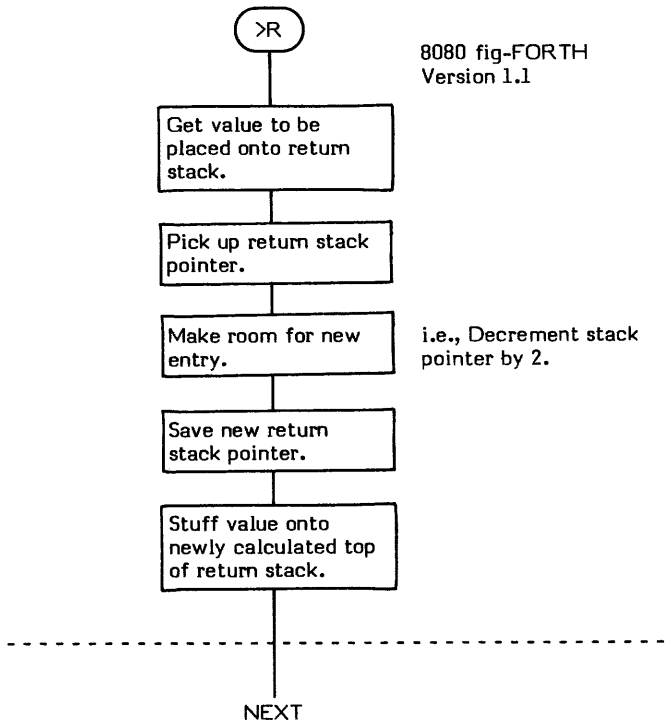
Note: Care must be taken to ensure that the return stack is restored to its original condition before returning to the calling procedure. This is usually accomplished through the use of R> (e.g., : NEW-WORD >R .... R> ;).

* **At entry** - The top of the parameter stack contains the 16-bit value to be placed onto the return stack.
* **At exit** - No parameter stack parameters. The top of the return stack contains the value previously on the top of the parameter stack.

>R is a low level code primitive.

Refer to R> .

FORTH-79: The FORTH-79 equivalent for >R is >R .



? (address of value to be output --)

? (pronounced "question-mark") performs a binary-to-ascii conversion on the signed 16-bit value contents of the specified memory location and prints the result on the output device. For example, a variable name followed by ? prints the contents of that variable.

The current value in BASE is used as the conversion radix.

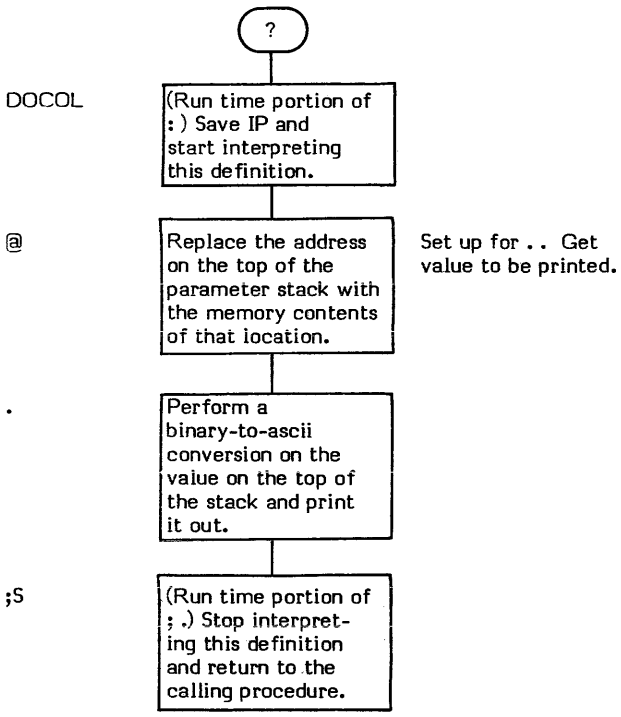
* **At entry** - The top of the parameter stack contains the memory address of the 16-bit signed value to be converted and printed.
* **At exit** - No parameters.

? is a high level colon definition.

Refer to D.R .

FORTH-79: The FORTH-79 equivalent for ? is ? .

Definition: : ? (address --)
@ . ;



?COMP

?COMP (--)

?COMP (pronounced "question-compile") issues Error Message 11H and a QUIT (8080 fig-FORTH Version 1.1) if the system is not in compile mode. In this case, compile mode is defined as the user variable STATE containing a non-zero value.

The standard error text for this word is "COMPILATION ONLY, USE IN DEFINITION".

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

?COMP is used by compiler words to ensure that the system is in compile mode. COMPILE and several of the conditional words are examples of words which use ?COMP .

* **At entry** - No parameters.
* **At exit** - No parameters.

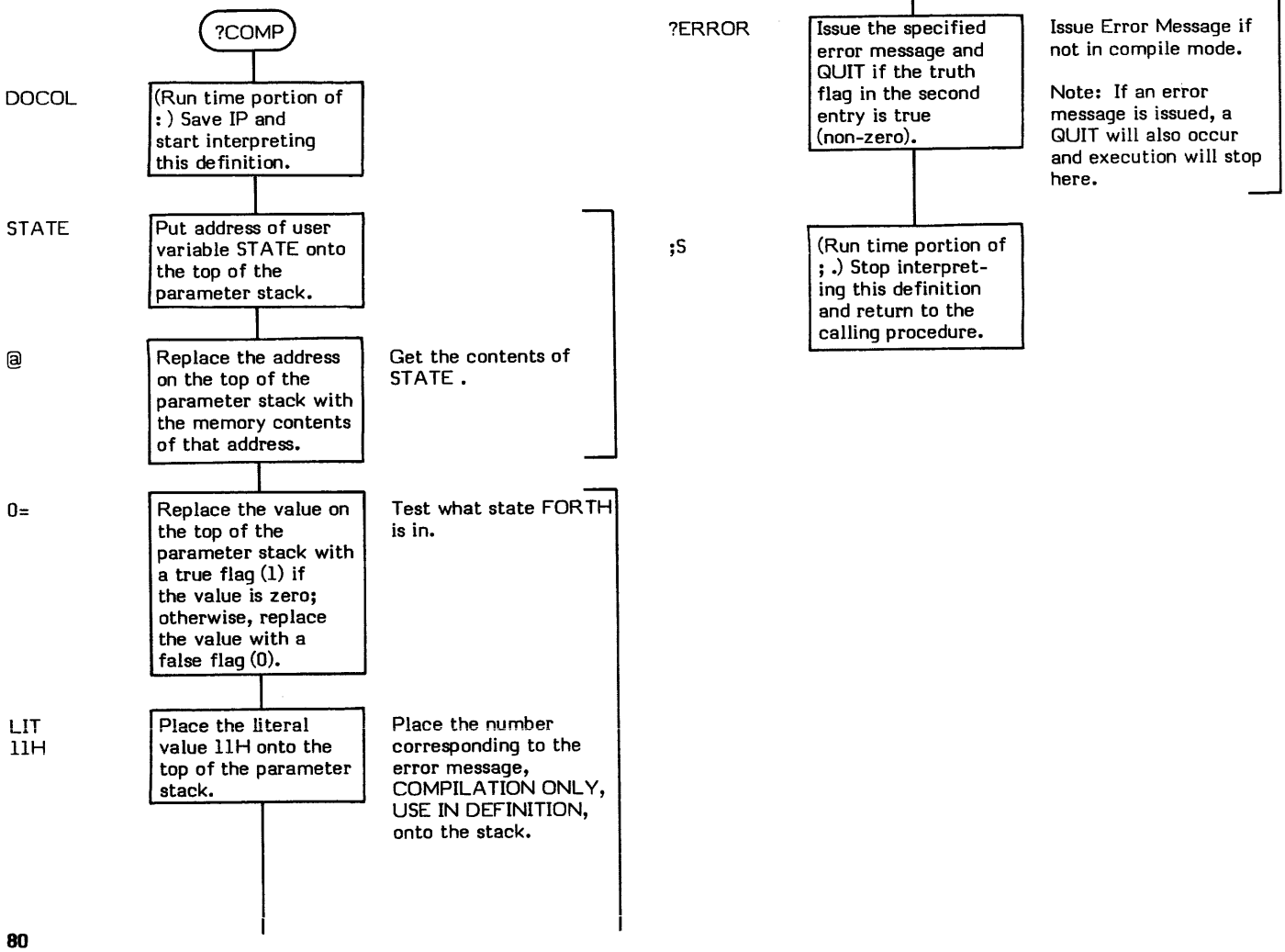
?COMP is a high level colon definition.

Refer to WARNING , MESSAGE , STATE , ERROR , (ABORT) , QUIT , and COMPILE .

FORTH-79: There is no FORTH-79 equivalent for ?COMP .

Definition:

```
: ?COMP ( -- )
    STATE @ 0= 11 ?ERROR ;
```



?CSP (--)

?CSP (pronounced "question-C-S-P") issues Error Message 14H and a QUIT if the current parameter stack pointer position does not equal that stored in the user variable CSP .

The standard error text for this word is "DEFINITION NOT FINISHED".

This is most often used in compiler security since an unbalanced stack often reflects a compilation error. Normally a !CSP is used to save the stack pointer position when beginning compilation (e.g., in :) and ?CSP is used to check it when finishing compiling.

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

An example of the use of ?COMP can be found in ;CODE .

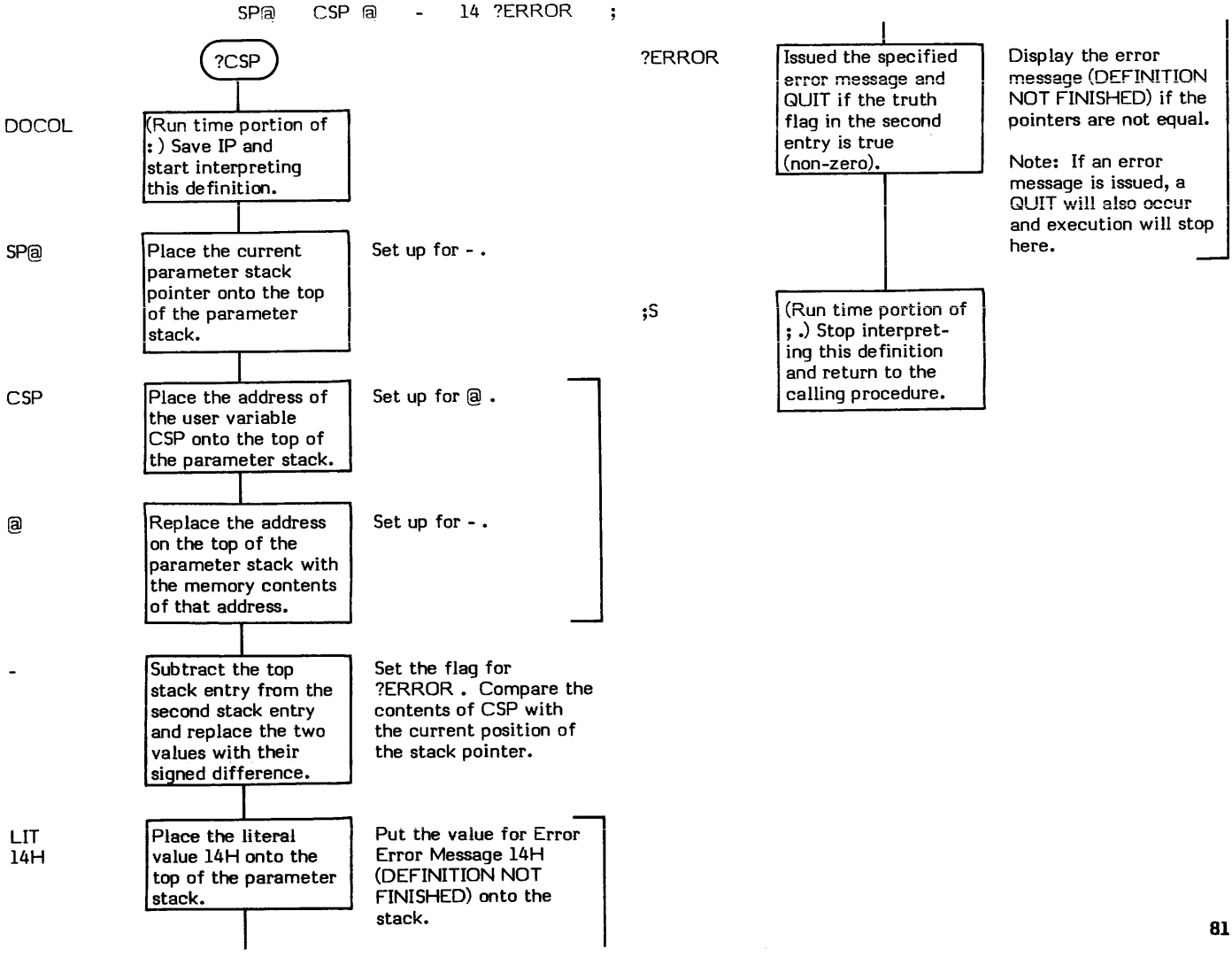
* **At entry** - No parameters.
* **At exit** - No parameters.

?CSP is a high level colon definition.

Refer to WARNING , MESSAGE , ?ERROR , STATE , (ABORT) , and QUIT .

FORTH-79: There is no FORTH-79 equivalent for ?CSP .

Definition: : ?CSP (--)
SP@ CSP @ - 14 ?ERROR ;



?ERROR

?ERROR (truth flag \ Error Message Number —)

?ERROR (pronounced "question-error") issues the specified error message and a QUIT if the truth flag is true (non-zero).

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

FORGET is an example of a word which uses ?ERROR.

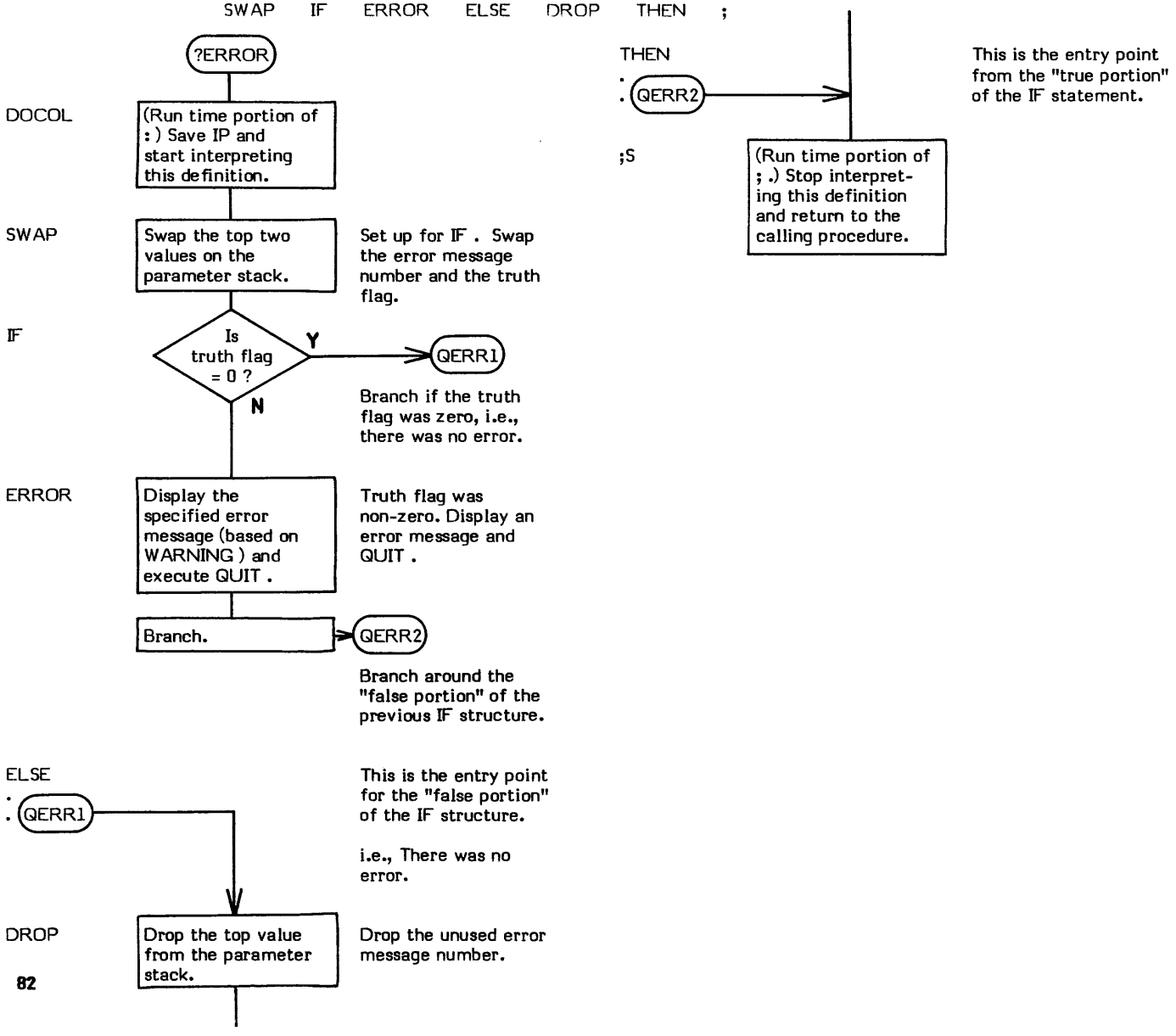
* **At entry** - The top of the parameter stack contains a signed 16-bit single precision message number value. The second stack entry contains a boolean truth flag.
* **At exit** - No parameters.

?ERROR is a high level colon definition.

Refer to ERROR , MESSAGE , WARNING , QUIT , ABORT , and (ABORT) .

FORTH-79: There is no FORTH-79 equivalent for ?ERROR .

Definition: : ?ERROR (flag \ error message number)
SWAP IF ERROR ELSE DROP THEN ;



?EXEC (–)

?EXEC (pronounced "question-execute") issues Error Message 12H and a QUIT if the system is not in execute mode. In this case, execution mode is defined as the user variable STATE containing a value of zero.

The standard error text for this word is "EXECUTION ONLY".

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

: (colon) is an example of a word which uses ?EXEC.

* **At entry** - No parameters.
* **At exit** - No parameters.

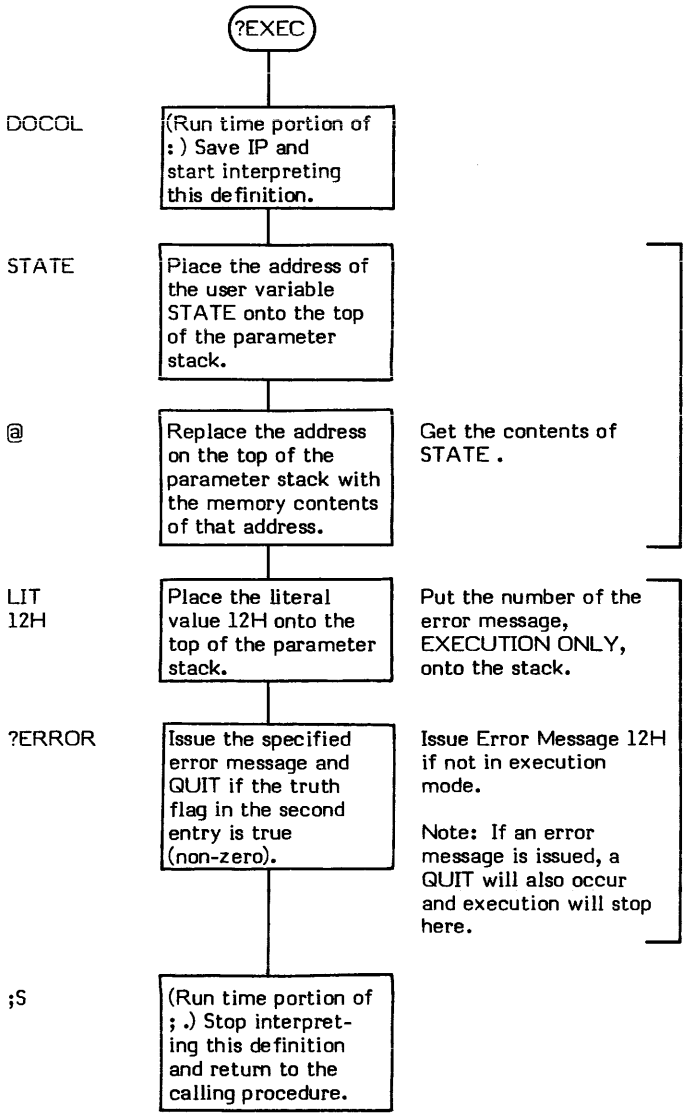
?EXEC is a high level colon definition.

Refer to WARNING , STATE , ERROR , (ABORT) , and QUIT .

FORTH-79: There is no FORTH-79 equivalent for ?EXEC .

Definition:

```
:  ?EXEC  ( -- )
      STATE @   12 ?ERROR  ;
```



?LOADING

?LOADING (–)

?LOADING (pronounced "question-loading") issues Error Message 16H and executes a QUIT if the system is not loading from disk.

The standard error text for the word is "USE ONLY WHEN LOADING".

The system normally receives input from disk (loading) or from the terminal keyboard. The user variable BLK contains a zero if the system is receiving input from the terminal input buffer. ?LOADING tests the contents of BLK and issues the error message if BLK contains a zero.

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

--> is an example of a word which uses ?LOADING .

* **At entry** - No parameters.
* **At exit** - No parameters.

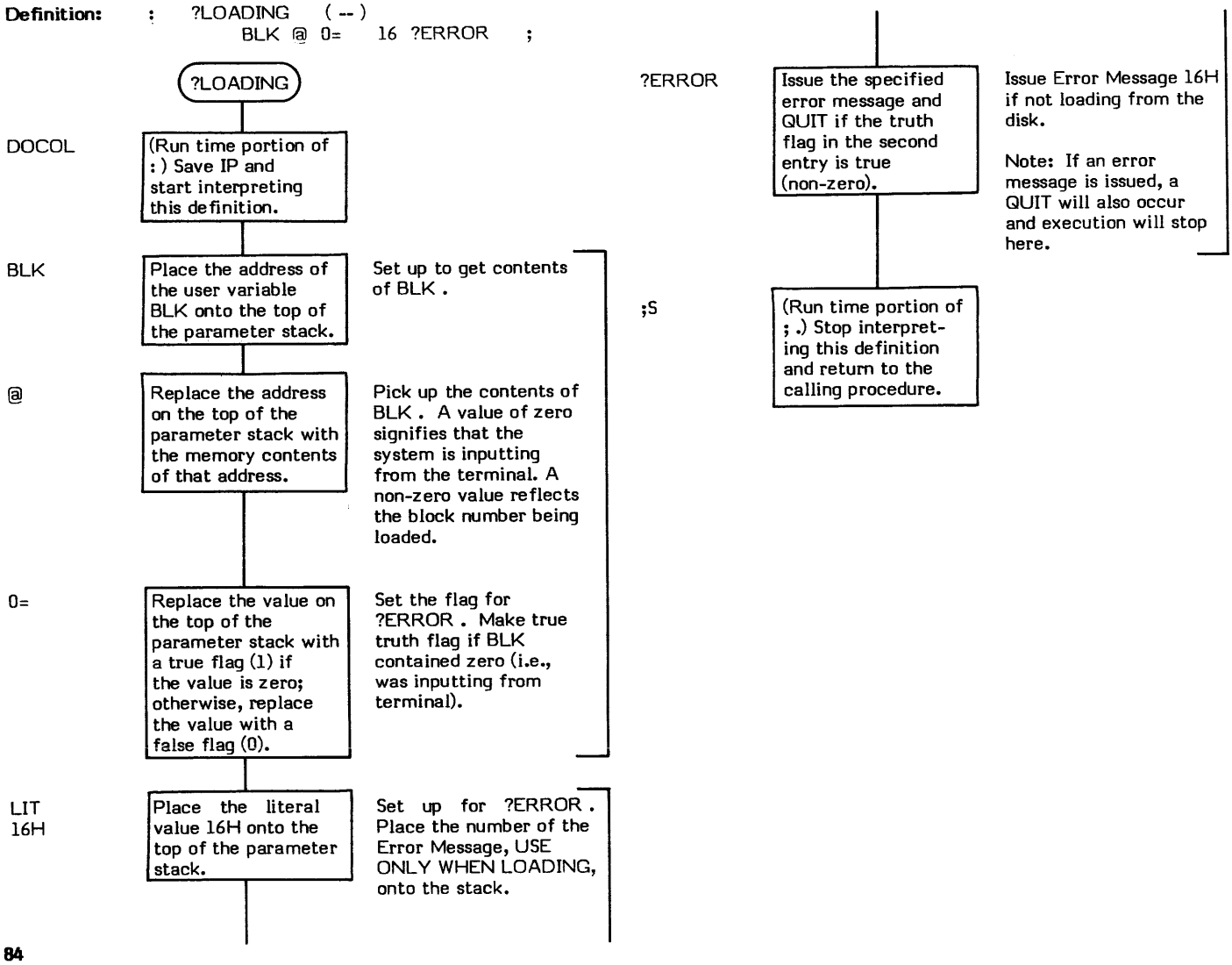
?LOADING is a high level colon definition.

Refer to ?ERROR , STATE , MESSAGE , WARNING , (ABORT) , and QUIT .

FORTH-79: There is no FORTH-79 equivalent for ?LOADING .

Definition:

```
:  ?LOADING  ( -- )
      BLK @ 0=   16 ?ERROR   ;
```



?PAIRS (value1 \value2 —)

?PAIRS (pronounced "question-pairs") issues Error Message 13H and executes a QUIT (8080 fig-FORTH Version 1.1) if the top two 16-bit values on the parameter stack are not equal.

The standard error text for this word is "CONDITIONALS NOT PAIRED".

This word is normally used during compilation to evaluate compilation conditionals.

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

AGAIN is an example of a word which uses ?PAIRS .

* **At entry** - The first and second parameter stack entries contain 16-bit values to be compared for an equal condition.
* **At exit** - No parameters.

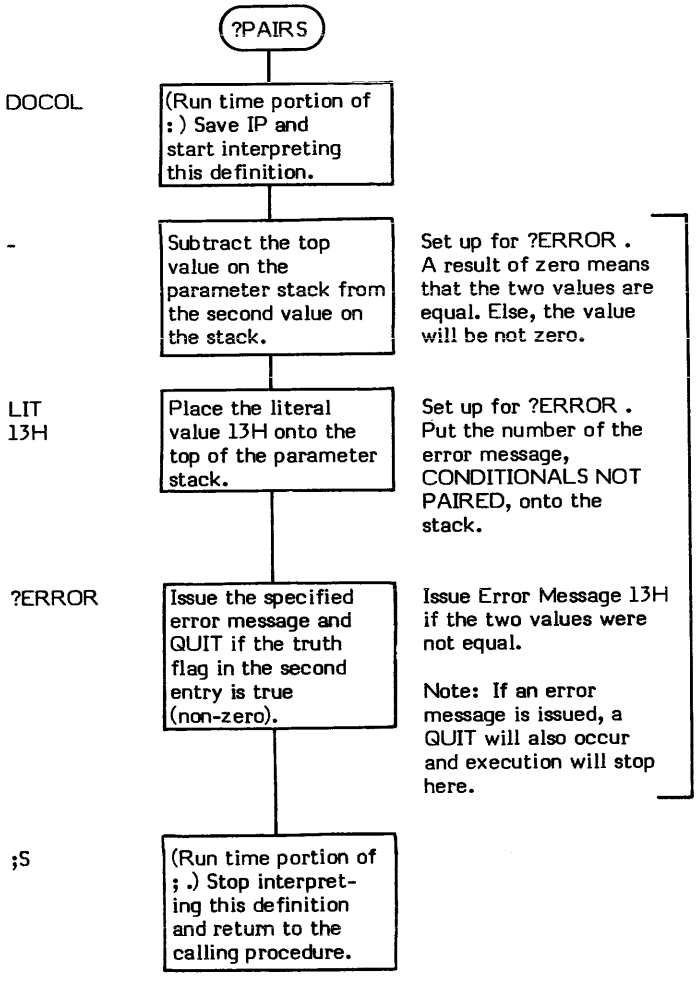
?PAIRS is a high level colon definition.

Refer to ?ERROR , WARNING , MESSAGE , STATE , (ABORT) , and QUIT .

FORTH-79: There is no FORTH-79 equivalent for ?PAIRS .

Definition:

```
: ?PAIRS ( value1 \value2 -- )
    - 13 ?ERROR ;
```



?STACK

?STACK (–)

?STACK (pronounced "question-stack") checks for stack underflow and stack overflow and executes ?ERROR if either condition has occurred.

Stack underflow generally occurs when the stack pointer "backed up" behind the initial starting location.

Stack overflow generally occurs when the stack has "grown" into the PAD area just below the end of the dictionary.

The exact determination of stack underflow and overflow is installation dependent which makes ?STACK an installation dependent word.

The standard error text for this word is "EMPTY STACK" and "STACK OVERFLOW".

The exact nature of the action taken when an error message is issued depends upon the contents of the user variable WARNING . (See WARNING and MESSAGE).

If WARNING contains a 0 value, only the error message number is output and then a QUIT is performed.

If WARNING contains a positive, non-zero value, the message number is used as an offset (plus or minus) relative to Line 0 of Screen 4. (e.g., A message number of 2 results in Line 2 of Screen 4 being displayed.) After the message is issued, a QUIT is performed.

If WARNING contains a negative value, normally -1, (ABORT) is executed. No message is output.

INTERPRET is an example of a word which uses ?STACK.

* **At entry** - No parameters.
* **At exit** - No parameters.

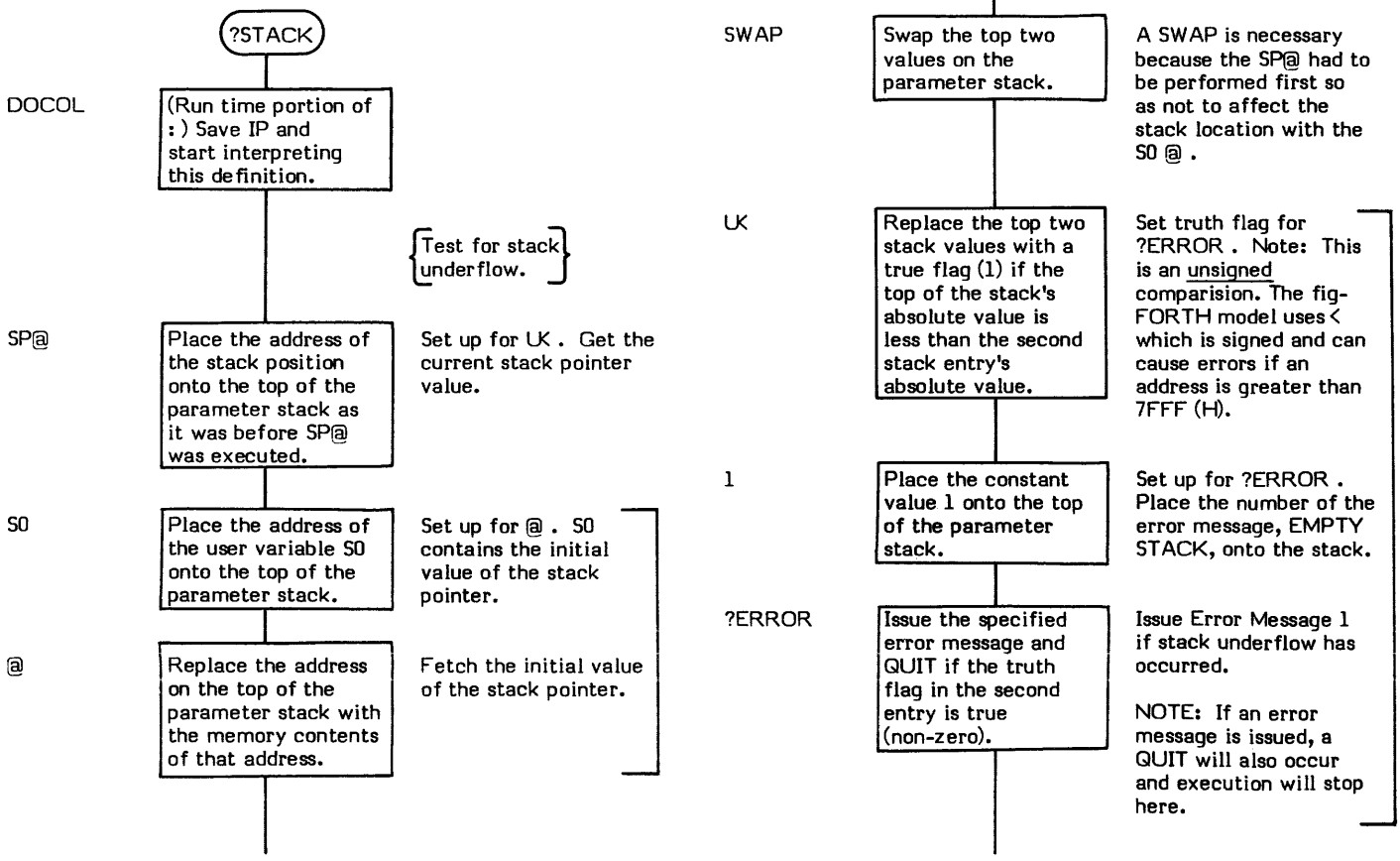
?STACK is a high level colon definition.

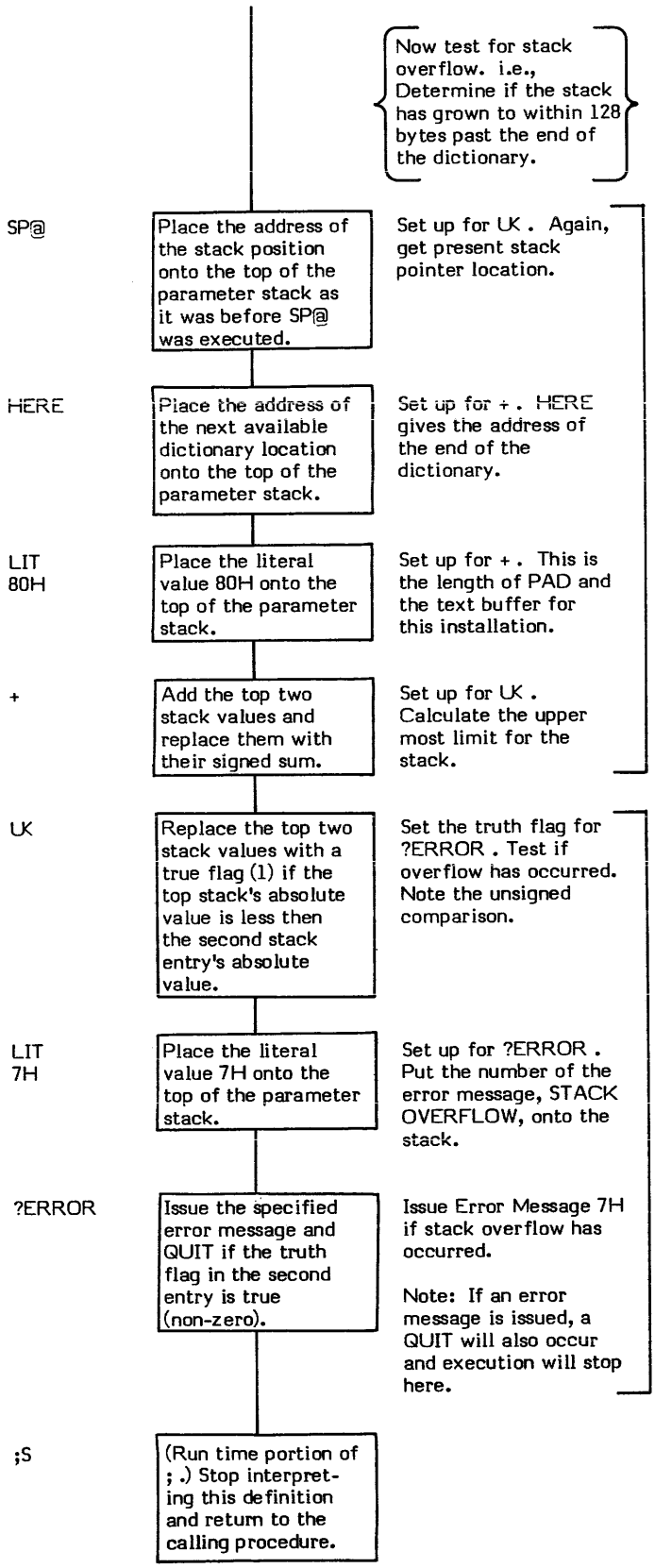
Refer to ERROR , WARNING , MESSAGE , QUIT , ABORT , and (ABORT) .

FORTH-79: There is no FORTH-79 equivalent for ?STACK .

Definition:

```
:  ?STACK  ( -- )
       SP@  S0 @  SWAP  UK 1 ?ERROR  SP@ HERE 80 +
       UK 7 ?ERROR  ;
```




Now test for stack overflow. i.e., Determine if the stack has grown to within 128 bytes past the end of the dictionary.
SP@
Place the address of the stack position onto the top of the parameter stack as it was before SP@ was executed.
Set up for UK . Again, get present stack pointer location.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
Set up for + . HERE gives the address of the end of the dictionary.
LIT 80H
Place the literal value 80H onto the top of the parameter stack.
Set up for + . This is the length of PAD and the text buffer for this installation.
+
Add the top two stack values and replace them with their signed sum.
Set up for UK . Calculate the upper most limit for the stack.
UK
Replace the top two stack values with a true flag (1) if the top stack's absolute value is less then the second stack entry's absolute value.
Set the truth flag for ?ERROR . Test if overflow has occurred. Note the unsigned comparison.
LIT 7H
Place the literal value 7H onto the top of the parameter stack.
Set up for ?ERROR . Put the number of the error message, STACK OVERFLOW, onto the stack.
?ERROR
Issue the specified error message and QUIT if the truth flag in the second entry is true (non-zero).
Issue Error Message 7H if stack overflow has occurred.
Note: If an error message is issued, a QUIT will also occur and execution will stop here.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

?TERMINAL

?TERMINAL (– flag)

?TERMINAL (pronounced "question-terminal") is an installation dependent word that is normally used to determine if there is a character ready to be input from a terminal.

This word is sometimes used to determine first if the break key has been pressed; then if any other key has been pressed.

INDEX is an example of a word which uses ?TERMINAL .

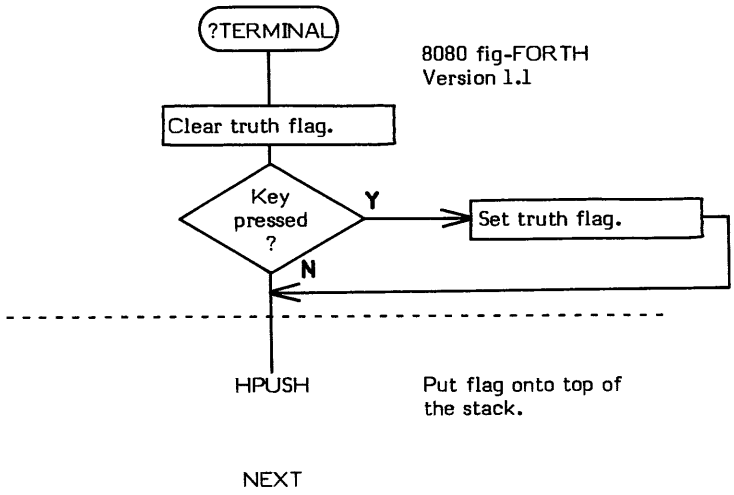
* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains a truth flag.

A true flag (1) indicates that a character is ready. KEY can then be used to read that character.

A false flag (0) indicates that no key has been pressed.

?TERMINAL is a low level code primitive.

FORTH-79: There is no FORTH-79 equivalent for ?TERMINAL .



@ (memory address — 16-bit memory contents)

@ (pronounced "fetch") replaces the 16-bit address on the top of the parameter stack with the 16-bit memory contents of that address.

@ is the primary means in FORTH of accessing data stored in memory.

NOTE: This word "fetches" word (16 bit) values; C@ is used to fetch byte (8 bit) values.

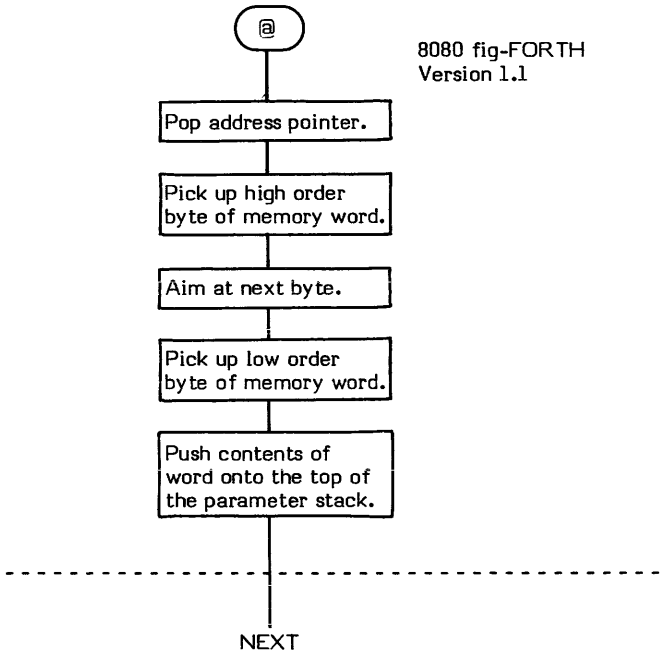
LOAD is an example of a word which uses @.

* **At entry** - The top of the parameter stack contains the 16-bit address of the memory word to be fetched.
* **At exit** - The top of the parameter stack contains the 16-bit contents of the specified memory word.

@ is a low level code primitive.

Refer to C@ .

FORTH-79: The FORTH-79 equivalent for @ is @ .



ABORT

ABORT (–)

ABORT can be considered the warm start routine for the FORTH system. ABORT resets the parameter stack, displays a start up message, sets CONTEXT and CURRENT to FORTH , sets BASE to DECIMAL , and issues a QUIT (which stops any compilation and starts interpretation from the input terminal).

ABORT is commonly called by (ABORT) when WARNING is -1 and an error condition has been detected.

COLD is an example of a word which uses ABORT .

* **At entry** - No parameters.
* **At exit** - No parameters.

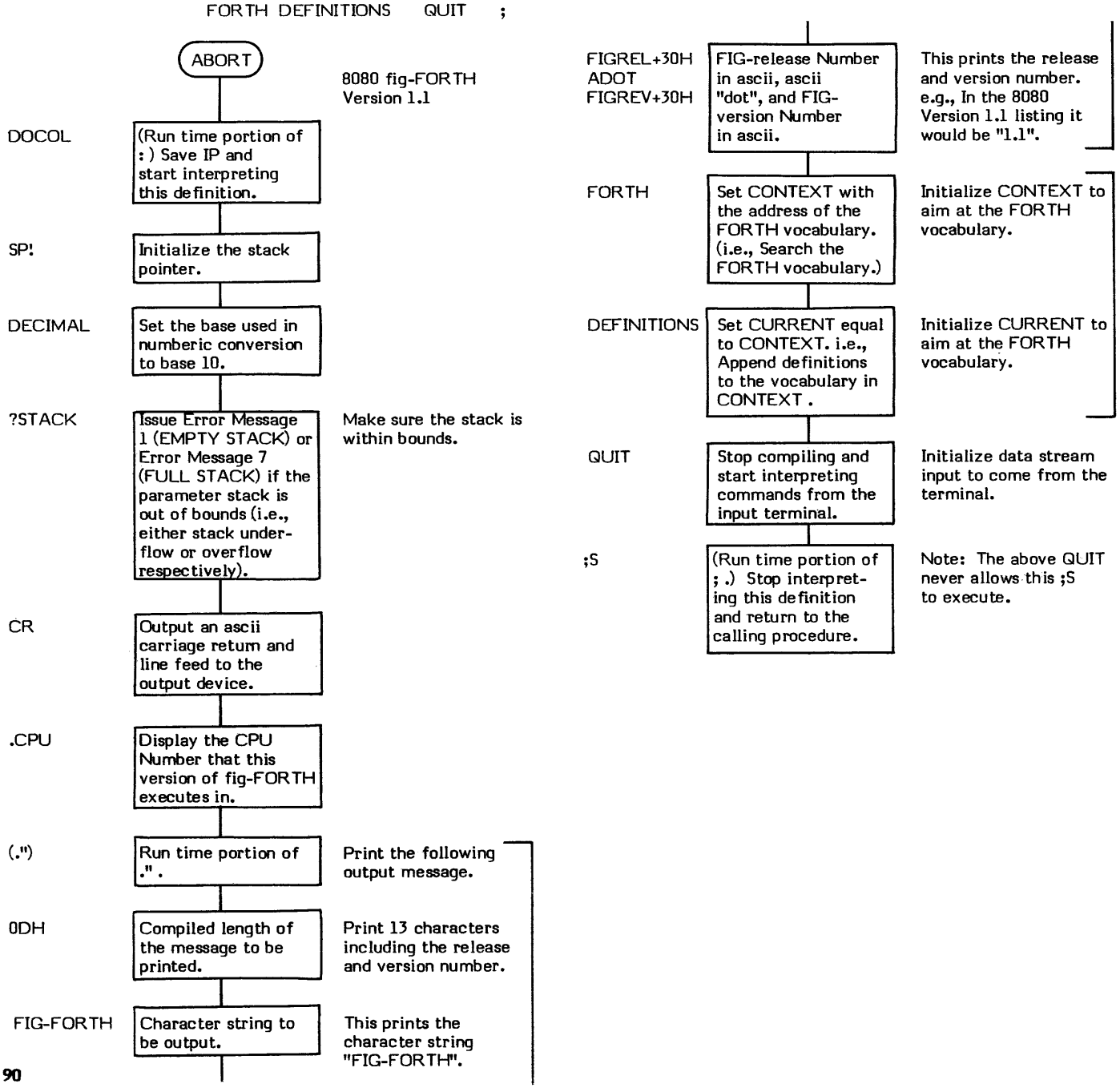
ABORT is a high level colon definition.

Refer to (ABORT) , COLD , and QUIT .

FORTH-79: The FORTH-79 equivalent for ABORT is ABORT .

Definition:

```
:  ABORT  ( – )
      SP! DECIMAL ?STACK CR .CPU  ." FIG-FORTH 8080 VER 1.1 "
      FORTH DEFINITIONS  QUIT  ;
```



ABS (signed value — absolute value)

ABS (pronounced "absolute") converts a signed single precision value on the top of the parameter stack into its absolute unsigned value.

M* is an example of a word that uses ABS .

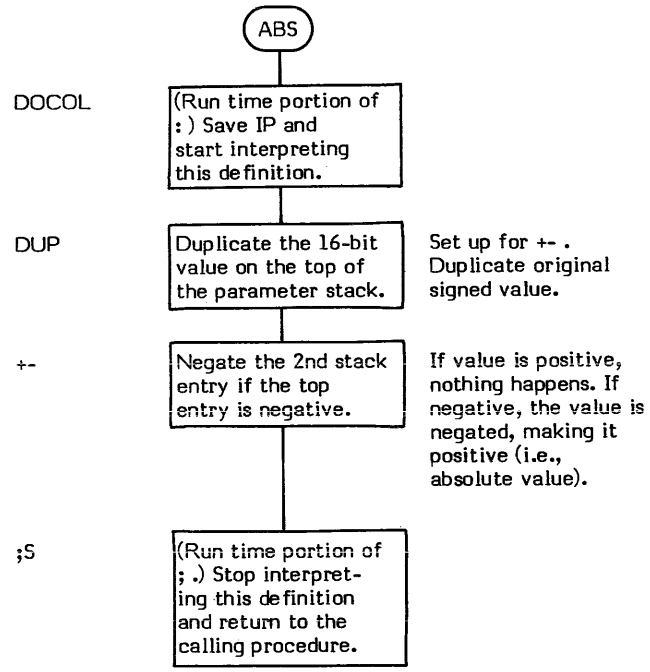
* **At entry** - The top of the parameter stack contains a 16-bit signed value.
* **At exit** - The top of the parameter stack contains a 16-bit unsigned absolute value.

ABS is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for ABS is ABS .

Definition:

```
:  ABS  ( signed value -- absolute value )
     DUP  +-  ;
```



AGAIN

AGAIN

COMPILE TIME: (loop address\ 1 --)
(Sequence 2)

EXECUTION TIME: (--)
(Sequence 3)

AGAIN is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

AGAIN is used to mark the end of an infinite loop structure in the form:

BEGIN "Loop Body" AGAIN

There is no exit from a BEGIN-AGAIN loop. If an exit is necessary, another type of loop structure should be used.

BEGIN-AGAIN loop structures must always be used within a colon definition.

The compile time action (Sequence 2) of AGAIN is to compile a BRANCH into the dictionary. Secondly, it resolves the loop body entry point address provided by BEGIN into a return branch offset used by BRANCH and stores this offset into the dictionary.

Some compiler security is provided by checking for a 1 on the top of the stack. BEGIN leaves a 1 on the stack. Since no other compiling words leave a 1 on the stack, failure to pair an AGAIN with a BEGIN will cause an error condition to be detected by ?PAIRS . Note that this is not foolproof.

The execution time action (Sequence 3) of AGAIN is to unconditionally branch back to its corresponding BEGIN (i.e., the beginning of the "loop body"). AGAIN accepts no input parameters and leaves nothing on the stack.

The BRANCH , compiled into the definition at compile time, is what performs the looping.

AGAIN may only be used within a colon (:) definition.

Note that AGAIN is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

QUIT is an example of a word which uses AGAIN.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 1 used for compiler security. The second stack entry contains the 16-bit entry point address of the "loop body" portion of the BEGIN-AGAIN structure.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

AGAIN is a high level colon definition.

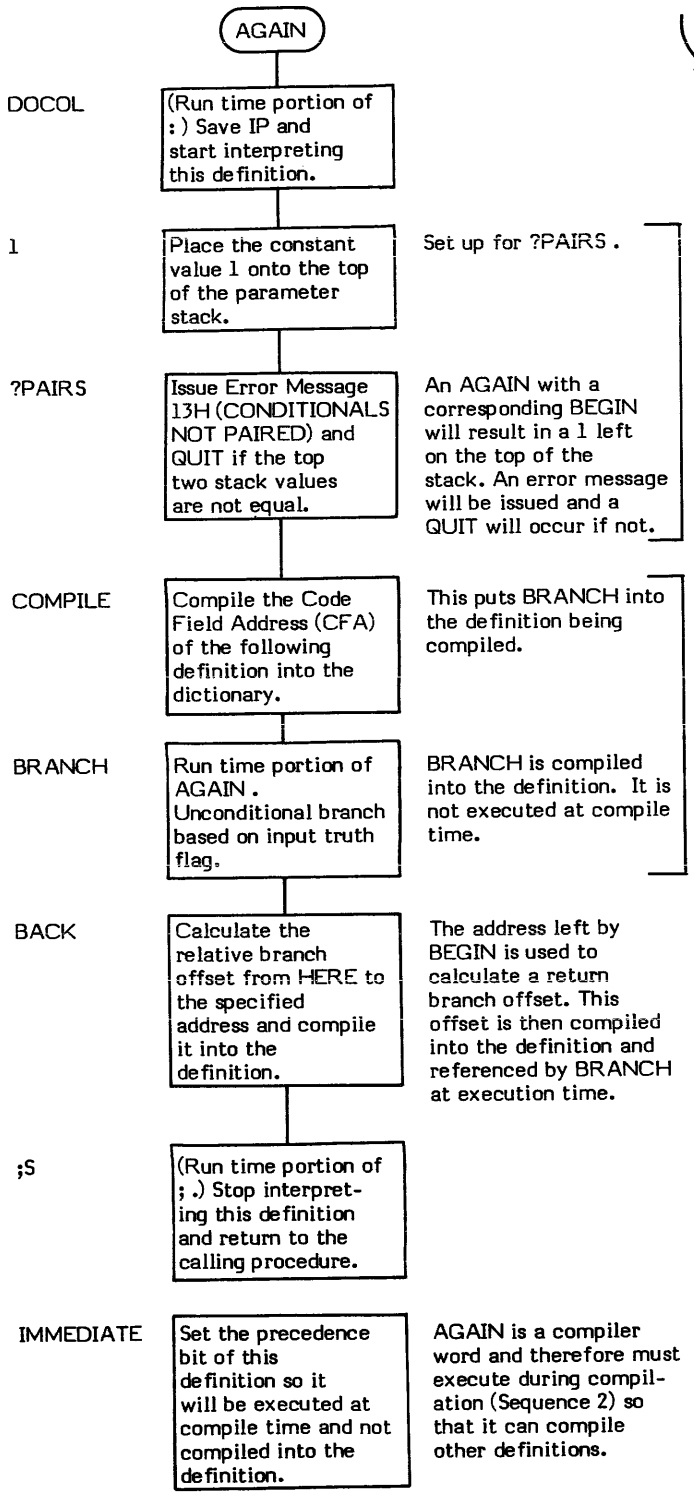
Refer to BEGIN , and BRANCH .

FORTH-79: AGAIN is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

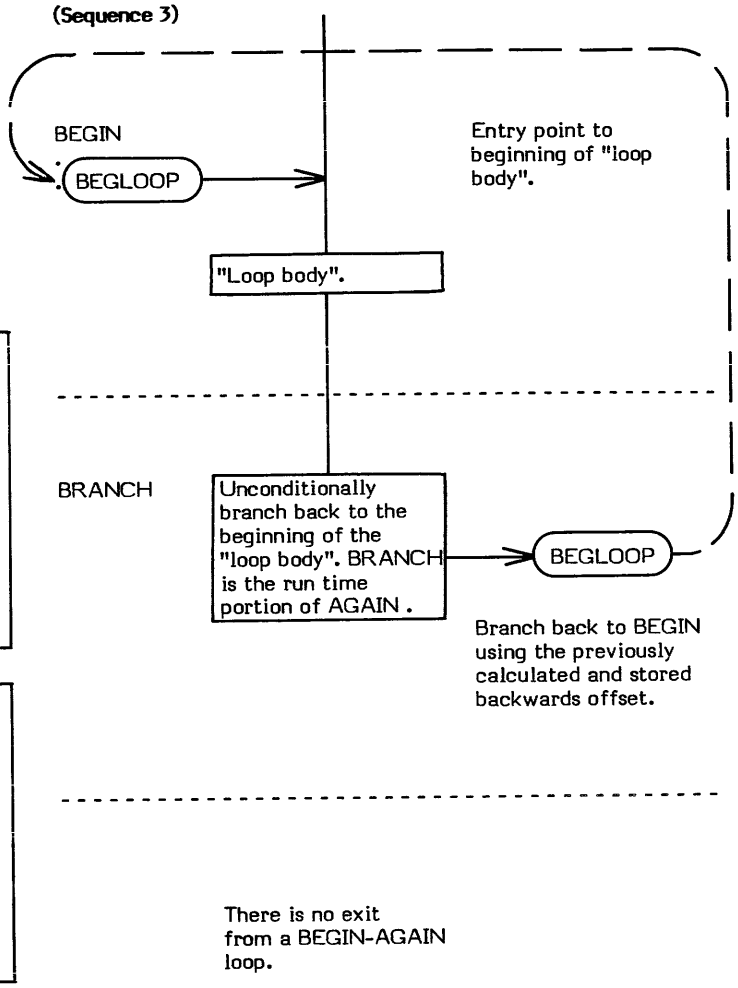
Definition:

```
:  AGAIN  ( loop address\1 -- )  ( compile time )
     1 ?PAIRS  COMPILE BRANCH  BACK  ;  IMMEDIATE
```

COMPILE TIME action of AGAIN : (loop address 1 —)
(Sequence 2)



EXECUTION TIME action of AGAIN : (—)
(Sequence 3)



ALLOT

ALLOT (number of storage locations --)

ALLOT advances the Dictionary Pointer (DP) the specified number of storage locations. Its primary purpose is to reserve or "allot" memory space in the dictionary. The particular length of a storage location is dependent upon the address type of the processor used.

." is an example of a word which uses ALLOT .

* **At entry** - The top of the parameter stack contains a 16-bit signed number specifying the number of storage locations to be allotted.

* **At exit** - No parameters.

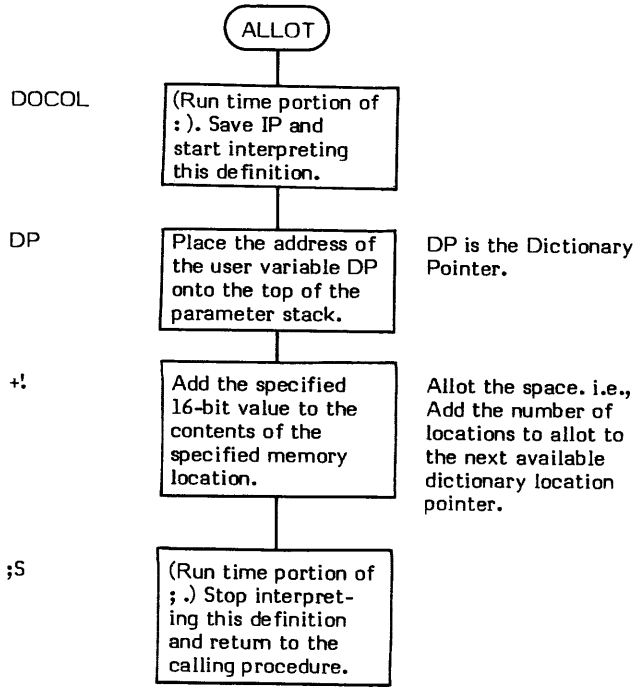
ALLOT is a high level colon definition.

Refer to DP .

FORTH-79: The FORTH-79 equivalent for ALLOT is ALLOT .

Definition:

```
:  ALLOT  ( number -- )
     DP +!  ;
```



AND (value1 \ value2 -- logical result)

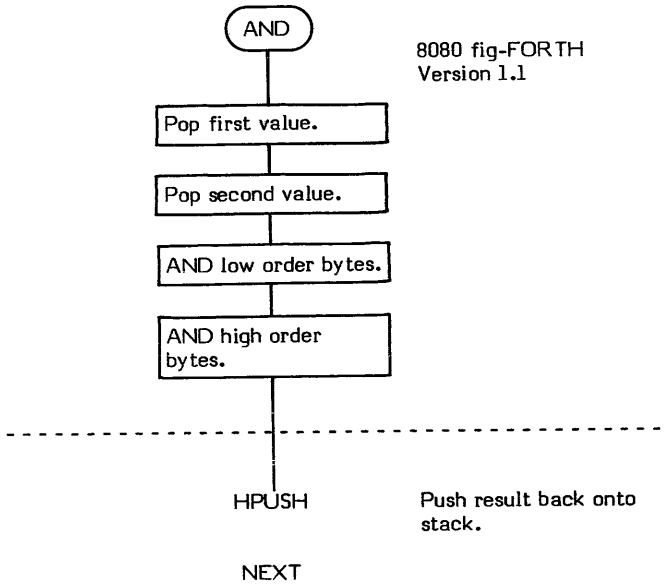
AND performs a bit-wise logical AND function on the top two values on the parameter stack and replaces them with their logical result.

BUFFER is an example of a word which uses AND.

* **At entry** - The top of stack and second entry contain the 16-bit values to be ANDed.
* **At exit** - The top of the stack contains the 16-bit logical result.

AND is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for AND is AND .



B/BUF

B/BUF (— bytes per buffer)

B/BUF (pronounced "bytes-per-buffer") is a single precision CONSTANT value. This constant places the number of bytes per disk buffer onto the top of the parameter stack. This value reflects the number of bytes transferred between mass-storage and memory by BLOCK .

The number of buffers in a system can be determined using B/BUF by subtracting FIRST from LIMIT and then dividing the result by B/BUF (i.e., LIMIT FIRST - B/BUF /).

(LINE) is an example of a word which uses B/BUF.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value of the number of bytes per disk buffer.

Refer to BUFFER , +BUF , BLOCK , LIMIT , and FIRST .

FORTH-79: B/BUF is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set". Note that the size of a FORTH-79 buffer is 1024 bytes (1K).

B/SCR (— blocks per screen)

B/SCR (pronounced "blocks-per-screen") is a single precision CONSTANT value. This constant places the number of blocks per editing screen onto the top of the parameter stack. By convention, an editing screen in FORTH contains 1024 (decimal) or 1K characters. The screen is normally organized into 16 lines of 64 characters. The constant C/L specifies how many characters are used per editing line.

MESSAGE is an example of a word which uses B/SCR.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value of the number of blocks per editing screen.

FORTH-79: There is no FORTH-79 equivalent for B/SCR .

BACK

BACK (entry point address —)

BACK is primarily used while compiling loop structure words at Sequence 2. It resolves the supplied address into a backwards (hence BACK) branch offset and compiles the offset into the dictionary.

For example, during compilation (Sequence 2), DO supplies the beginning address of the "loop body" to LOOP . BACK (contained within LOOP) converts this address into an offset and then compiles it for (LOOP) to use when "branching back" during execution time (Sequence 3).

LOOP is an example of a word which uses BACK.

* **At entry** - The top of the parameter stack contains a 16-bit entry point address.
* **At exit** -No parameters.

BACK is a high level colon definition.

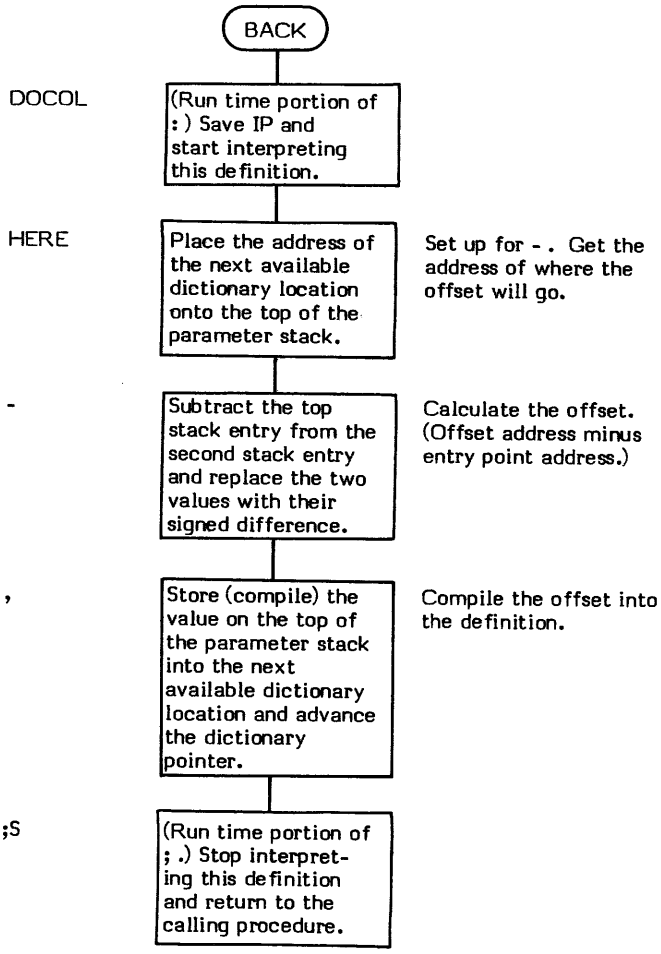
NOTE: Although BACK is a word used by compiler words, it is not an IMMEDIATE word itself.

Refer to DO , and LOOP .

FORTH-79: There is no FORTH-79 equivalent for BACK .

Definition:

```
: BACK ( entry point address -- )
    HERE - , ;
```



BASE (-- data address)

BASE is a user variable that contains the current number base (or radix) used for input and output numeric conversion.

HEX and DECIMAL are examples of words that set BASE . Executing HEX stores a 16 (decimal) into BASE . Executing DECIMAL stores a 10 (decimal) into BASE .

. ("dot") references BASE when performing a binary-ascii conversion.

(NUMBER) references BASE when performing ascii-binary conversion.

Note that the sequence BASE ? will always result in a "10" being output no matter what base the system is in. Since this is the correct answer, use the sequence BASE @ 1 - . which will print a "9" for a DECIMAL base, an "F" for a HEX base, etc. Another alternative is defining the word BASE? . : BASE? BASE @ DUP DECIMAL . BASE ! ; This word will display the base in decimal without destroying the contents of BASE .

The user variable BASE is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable BASE .

Refer to HEX , DECIMAL , (NUMBER) , . , and USER .

FORTH-79: The FORTH-79 equivalent for BASE is BASE .

BEGIN

BEGIN

COMPILE TIME: (— entry point address \ 1)
(Sequence 2)

EXECUTION TIME: (—)
(Sequence 3)

BEGIN is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

BEGIN is used to mark the beginning of an indefinite, repetitive loop structure. It is used as the beginning of three types of loops in the forms:

BEGIN "Loop Body" UNTIL

BEGIN "Loop Body" AGAIN

BEGIN "Set Conditional" WHILE "True Portion" REPEAT

A BEGIN-type loop structure must always be used within a colon definition.

BEGIN-type loops differ from DO-LOOP 's in that a BEGIN loop is not limited from its start to a specified number of passes through the loop as is a DO-LOOP . Once started, a BEGIN-type loop may execute indefinitely; or for some structures, until a specified condition is met.

The compile time action (Sequence 2) of BEGIN is very simple. BEGIN places the address of the next available dictionary location onto the parameter stack. (i.e., It passes the beginning address of the "loop body" to UNTIL , REPEAT or AGAIN so that these words can compile a return branch into their respective definitions.)

To provide compiler security, the value 1 is placed onto the top of the parameter stack so that UNTIL , AGAIN , WHILE or REPEAT can check for it. This provides a somewhat secure (but not foolproof) method of checking for un-balanced loop structures.

The apparent run time action (Sequence 3) of BEGIN is to serve as a return entry point for the "loop back" word at the end of the structure. (Really BEGIN does nothing because the branch which is executed is actually imbedded within the "loop back" definition.)

Note that BEGIN is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

QUIT is an example of a word which uses BEGIN.

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit signed single precision value 1 used for compiler security. The second stack entry contains a 16-bit address specifying the location of the first word of the "loop body".

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

BEGIN is a high level colon definition.

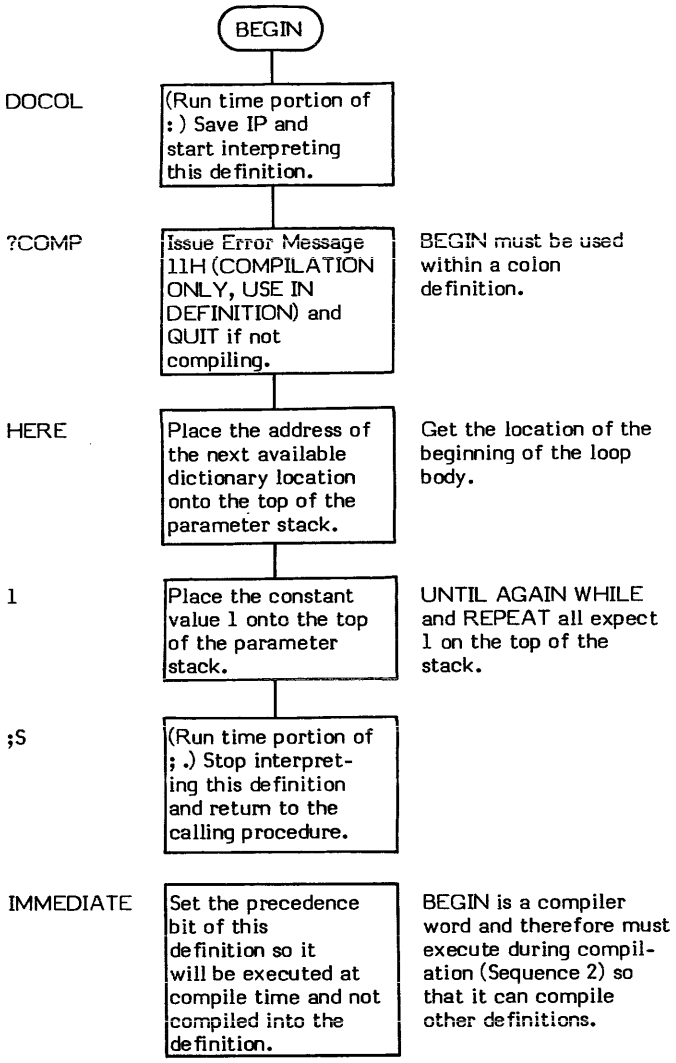
Refer to UNTIL , AGAIN , WHILE , and REPEAT .

FORTH-79: The FORTH-79 equivalent for BEGIN is BEGIN .

Definition:

```
:  BEGIN  ( -- entry point address \1 )  ( compile time )
       ?COMP  HERE  1  ;  IMMEDIATE
```

COMPILE TIME action of BEGIN (Sequence 2): (— entry point address \ 1)



EXECUTION TIME action of BEGIN (Sequence 3): (—)

There is no run time action for BEGIN .

BL

BL (— 20H)

BL (pronounced "B-L") is a single precision CONSTANT value. This constant places the ascii value for the character "blank" (or "space") onto the top of the parameter stack. This value is a 20 (hex) or a 32 (decimal). Referencing a blank via the name BL makes FORTH more readable.

BL is commonly used as a delimiter parameter for WORD . This not only makes the definition more readable, (e.g., BL WORD), but it allows less chance of introducing a "wrong base" bug. For example, if the editing screen is in decimal base, a blank is a 32; if the base is hex, a blank is a 20. This is sometimes a source of error.

-FIND is an example of a word which uses BL.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value of an ascii blank character.

Refer to WORD .

FORTH-79: There is no FORTH-79 equivalent for BL .

BLANKS (beginning address \ # of bytes to blank —)

BLANKS clears a specified region of memory to ascii blanks (20H).

BLANKS is simply a FILL with the fill character (20H) "hard coded".

* **At entry** - The top of the parameter stack contains the beginning address of the memory to fill. The second stack entry contains the positive 16-bit number of bytes (8080 fig-FORTH version) to fill.

* **At exit** - No parameters.

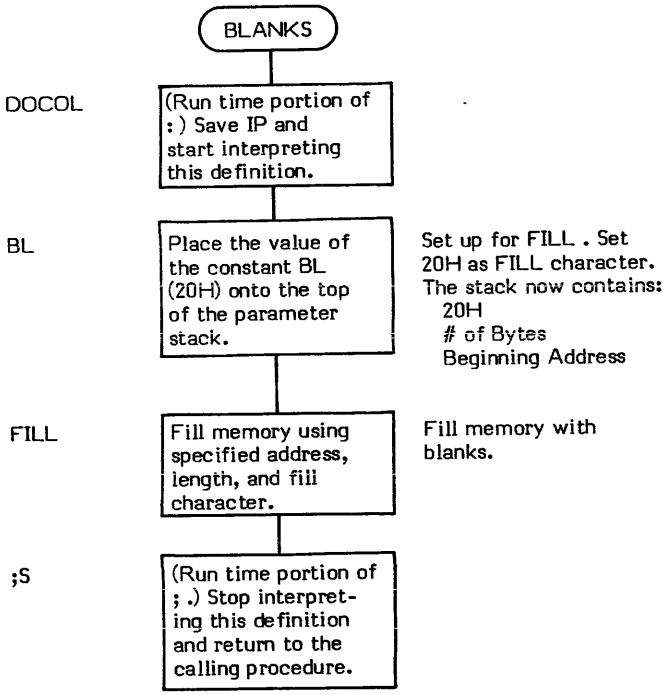
BLANKS is a high level colon definition.

Refer to FILL .

FORTH-79: BLANKS is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:

```
:  BLANKS  ( beginning address\# -- )
     BL  FILL  ;
```



BLK

BLK (-- data address)

BLK (pronounced "B-L-K") is a user variable which contains the number of the block currently being interpreted. If BLK contains a zero, the input data stream is from the terminal.

WORD references BLK to determine the address of the input data stream.

LOAD and --> set BLK with the mass storage block being transferred to memory.

QUIT sets BLK to 0, meaning that input is from the terminal.

The user variable BLK is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable BLK .

Refer to WORD , LOAD , --> , and USER .

FORTH-79: The FORTH-79 equivalent for BLK is BLK .

BLOCK (desired block number -- data address of desired block)

BLOCK is used to access data kept in mass storage. (Since disk is the most common mass storage, further references will be to "disk" and not "mass storage".) BLOCK is a virtual memory type access word. This means that the desired block number is replaced with the memory address of that block. The type of storage, the device number and other physical attributes may be treated as being transparent to the external operation of the word. This also means that data that has been flagged as modified, but which still resides in memory, is written back to disk before that location is overwritten with new disk data.

BLOCK transfers data from disk to a buffer. A FORTH system normally contains several physically contiguous buffers, with the last buffer "linked" by software (see +BUF) back to the first buffer to form a "logically circular" array of buffers. (Refer to +BUF for a more complete description of these buffers.)

The management of these buffers falls into two separate (but related) categories; buffer-referencing and buffer-allocation/disk access.

The function of buffer-referencing management is to minimize disk accesses. BLOCK 's primary responsibility is to perform buffer-referencing management. If buffer-allocation is necessary, BLOCK calls BUFFER (see BUFFER).

The basis of buffer-referencing management is to possibly save a disk access by first examining all buffers in the buffer array to determine if the desired block is currently in memory. The search begins with the most recently referenced buffer (pointed to by the system variable PREV), and proceeds cyclically through the buffer array until, if no match is found, it returns again to the most recently referenced buffer. Note that although the search takes place sequentially in the direction of buffer-allocation, there is no guarantee that the search takes place in descending order of the most recently referenced to least recently referenced buffer. This is because there is only one referencing-pointer (PREV) and references to blocks in memory buffers may take place in any random order.

Buffer-referencing management works because references to blocks are usually concentrated within the same set (but not necessarily numerically close range) of block numbers.

As previously described, if a newly allocated buffer has been flagged as being modified (see UPDATE); the contents of that buffer is written to disk and not just overlayed with the new data. An updated buffer may be forced to disk before this allocation process by using the word FLUSH .

It is extremely important to note that the size of a buffer (i.e., the number of bytes read by BLOCK) is an arbitrarily set value. This value is stored in the constant B/BUF . This means that while the buffer size may correspond to the sector size of the disk being used, it does not have to. It also means that in a fig-FORTH system, BLOCK does not automatically read in a 1K byte source screen. (This is a possible source of confusion to those switching between fig-FORTH and some FORTH, Inc., type systems. i.e. In fig-FORTH, a BLOCK is not necessarily synonymous with a screen.)

The word R/W performs the physical device selection and data transfers. Each storage device in the system has a unique range of block numbers assigned to it. This allows each block in the system to be unique. It is also in keeping with the virtual aspect of BLOCK in that device selection is based solely upon block number range. (e.g., Drive 0 may contain blocks 1 through 799; while Drive 2 may contain blocks 800 through 1199, etc.)

This explicit addressing does pose a slight problem, however. For example; it is difficult to remember that certain data on a diskette may reside at block 160 when the diskette is in Drive 1; but that, when the diskette is in Drive 2, that same data on the same diskette may reside at block 560.

BLOCK solves this problem through the use of the user variable OFFSET . Immediately upon entry, the contents of OFFSET is added to the desired block number. This allows implicit drive selection by previously setting (i.e., biasing) OFFSET to the starting block number of the selected drive. (See DR0 and DR1.) Explicit block addressing may be performed by ensuring that OFFSET contains the value 0 before calling BLOCK .

(LINE) is an example of a word which uses BLOCK.

* **At entry** - The top of the parameter stack contains the 16-bit value specifying the desired block number. The maximum block number range is from 0 to 32767. The most-significant bit must be 0 as this is used as the "update" flag.

* **At exit** - The top of the parameter stack contains the 16-bit memory address of the data portion of the desired block.

BLOCK is a high level colon definition.

Refer to BUFFER , B/BUF , UPDATE , FLUSH , OFFSET , DR0 , DR1 , and R/W .

FORTH-79: The FORTH-79 equivalent for BLOCK is BLOCK .

Definition:

```
:  BLOCK  ( block number -- address )
      OFFSET @ +   >R   PREV @   DUP @   R - DUP +
      IF
         BEGIN
            +BUF 0=   IF   DROP   R BUFFER   DUP R 1 R/W   2 -   THEN
            DUP @   R -   DUP + 0=
         UNTIL
         DUP PREV !
      THEN   R> DROP   2+   ;
```

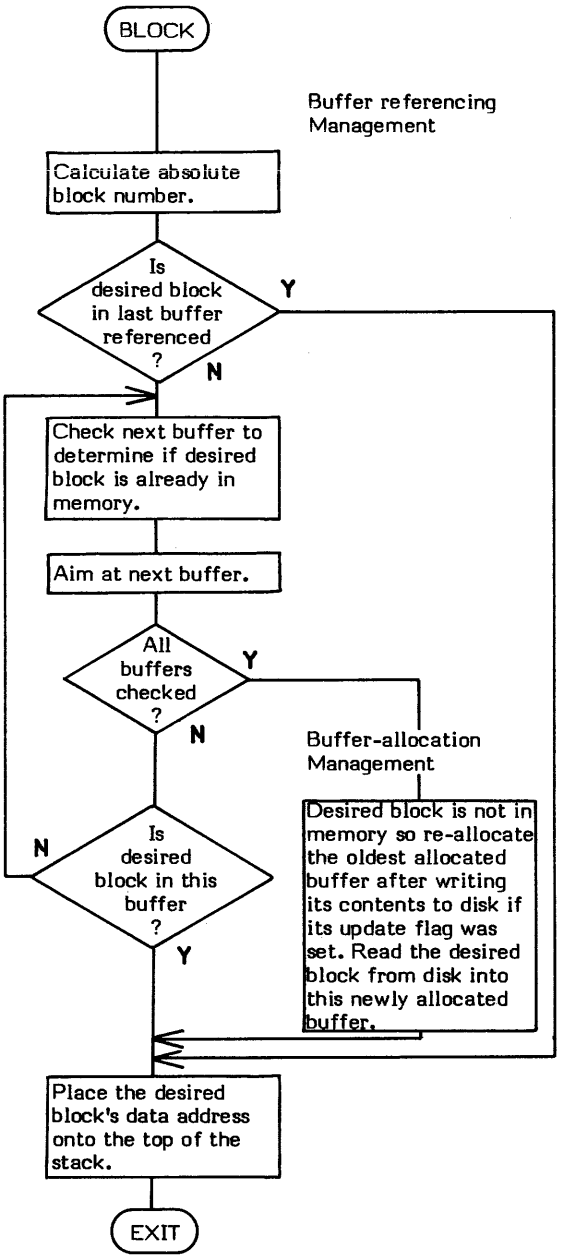
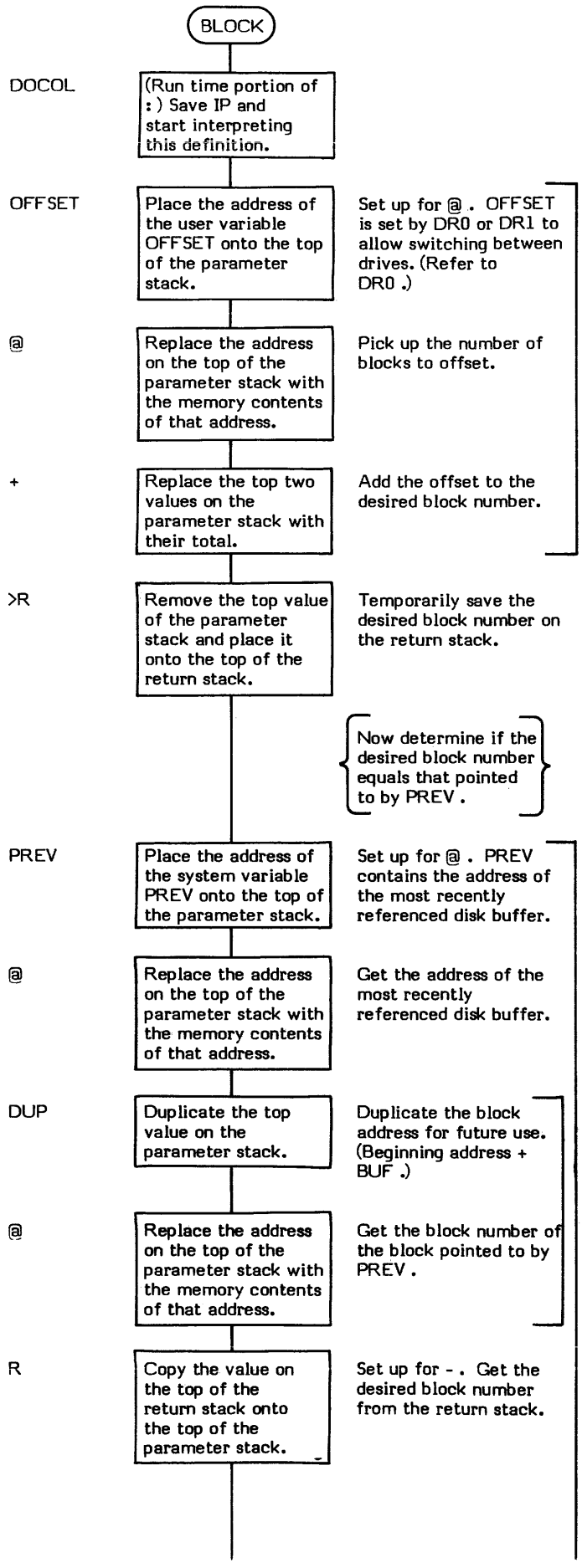
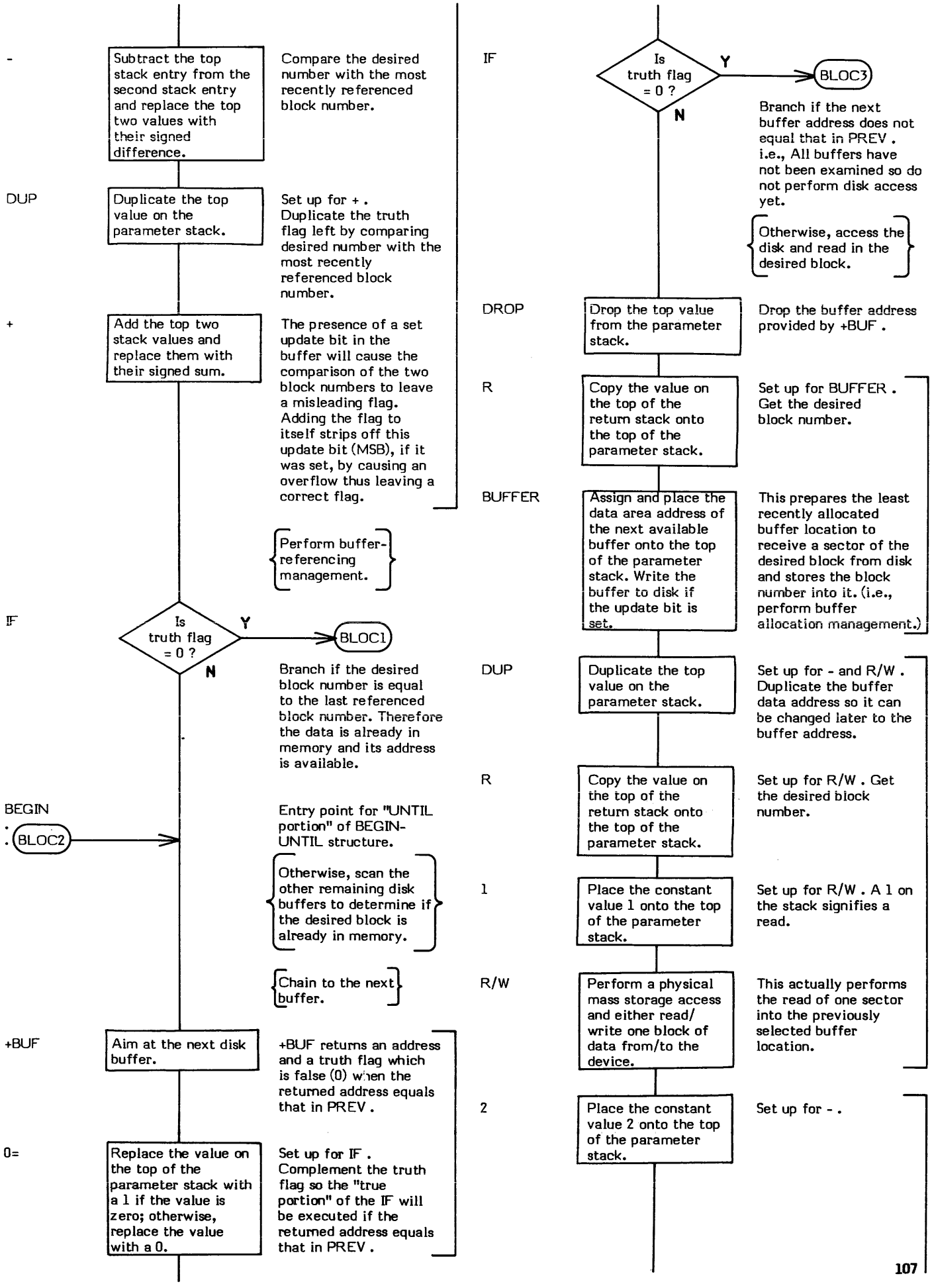
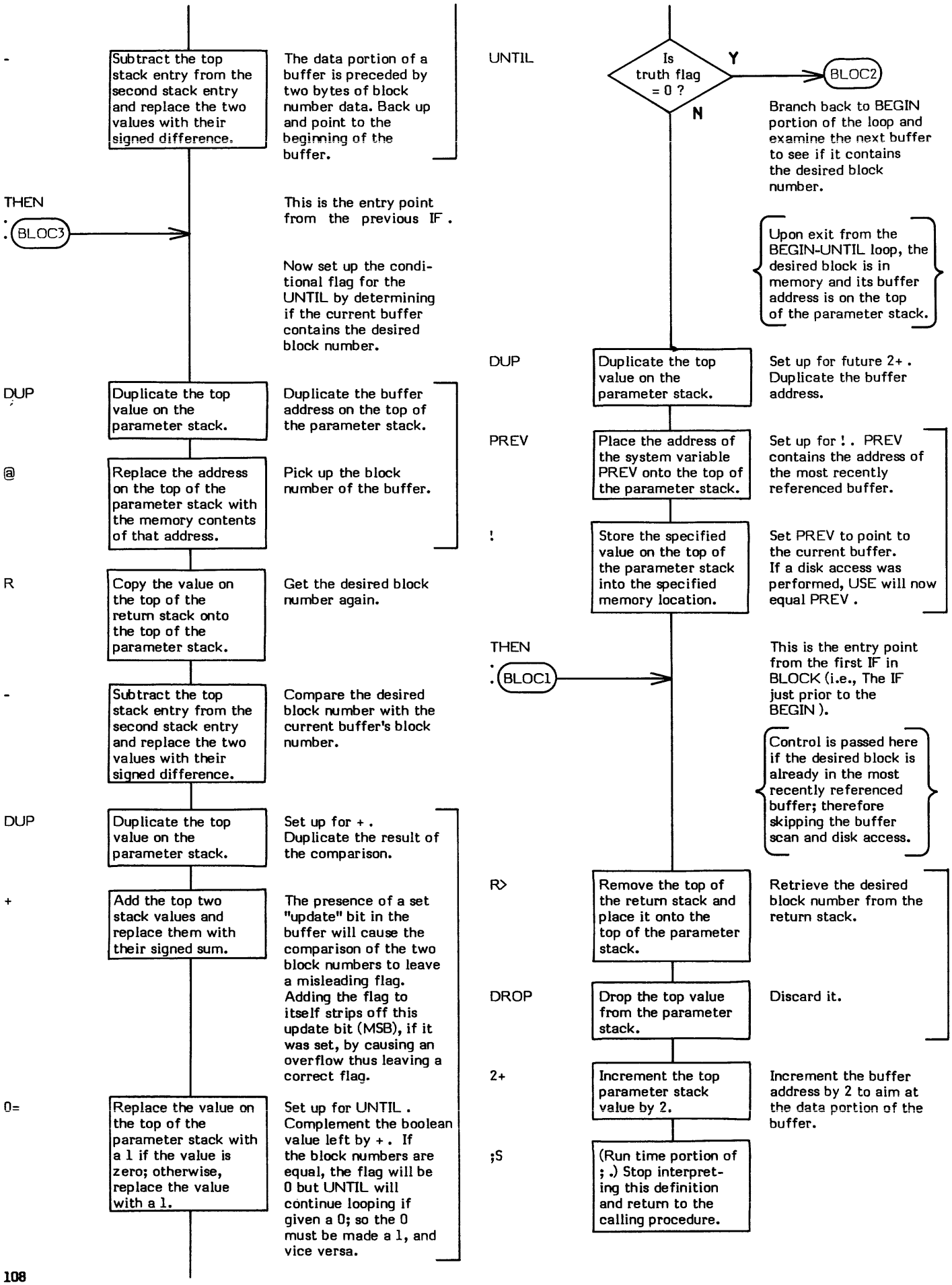


Figure BLOCK-1

High Level Flowchart of BLOCK




-
Subtract the top stack entry from the second stack entry and replace the top two values with their signed difference.
Compare the desired number with the most recently referenced block number.
DUP
Duplicate the top value on the parameter stack.
Set up for + . Duplicate the truth flag left by comparing desired number with the most recently referenced block number.
+
Add the top two stack values and replace them with their signed sum.
The presence of a set update bit in the buffer will cause the comparison of the two block numbers to leave a misleading flag. Adding the flag to itself strips off this update bit (MSB), if it was set, by causing an overflow thus leaving a correct flag.
Perform buffer-referencing management.
IF
Is truth flag = 0 ?
Y
N
BLOC1
Branch if the desired block number is equal to the last referenced block number. Therefore the data is already in memory and its address is available.
BEGIN
BLOC2
Entry point for "UNTIL portion" of BEGIN-UNTIL structure.
Otherwise, scan the other remaining disk buffers to determine if the desired block is already in memory.
Chain to the next buffer.
+BUF
Aim at the next disk buffer.
+BUF returns an address and a truth flag which is false (0) when the returned address equals that in PREV .
0=
Replace the value on the top of the parameter stack with a 1 if the value is zero; otherwise, replace the value with a 0.
Set up for IF . Complement the truth flag so the "true portion" of the IF will be executed if the returned address equals that in PREV .
IF
Is truth flag = 0 ?
Y
N
BLOC3
Branch if the next buffer address does not equal that in PREV . i.e., All buffers have not been examined so do not perform disk access yet.
Otherwise, access the disk and read in the desired block.
DROP
Drop the top value from the parameter stack.
Drop the buffer address provided by +BUF .
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for BUFFER . Get the desired block number.
BUFFER
Assign and place the data area address of the next available buffer onto the top of the parameter stack. Write the buffer to disk if the update bit is set.
This prepares the least recently allocated buffer location to receive a sector of the desired block from disk and stores the block number into it. (i.e., perform buffer allocation management.)
DUP
Duplicate the top value on the parameter stack.
Set up for - and R/W . Duplicate the buffer data address so it can be changed later to the buffer address.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for R/W . Get the desired block number.
1
Place the constant value 1 onto the top of the parameter stack.
Set up for R/W . A 1 on the stack signifies a read.
R/W
Perform a physical mass storage access and either read/write one block of data from/to the device.
This actually performs the read of one sector into the previously selected buffer location.
2
Place the constant value 2 onto the top of the parameter stack.
Set up for - .


-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
The data portion of a buffer is preceded by two bytes of block number data. Back up and point to the beginning of the buffer.
THEN
BLOC3
This is the entry point from the previous IF .
Now set up the conditional flag for the UNTIL by determining if the current buffer contains the desired block number.
DUP
Duplicate the top value on the parameter stack.
Duplicate the buffer address on the top of the parameter stack.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the block number of the buffer.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Get the desired block number again.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Compare the desired block number with the current buffer's block number.
DUP
Duplicate the top value on the parameter stack.
Set up for + . Duplicate the result of the comparison.
+
Add the top two stack values and replace them with their signed sum.
The presence of a set "update" bit in the buffer will cause the comparison of the two block numbers to leave a misleading flag. Adding the flag to itself strips off this update bit (MSB), if it was set, by causing an overflow thus leaving a correct flag.
0=
Replace the value on the top of the parameter stack with a 1 if the value is zero; otherwise, replace the value with a 1.
Set up for UNTIL . Complement the boolean value left by + . If the block numbers are equal, the flag will be 0 but UNTIL will continue looping if given a 0; so the 0 must be made a 1, and vice versa.
UNTIL
Is truth flag = 0 ?
Y
N
BLOC2
Branch back to BEGIN portion of the loop and examine the next buffer to see if it contains the desired block number.
Upon exit from the BEGIN-UNTIL loop, the desired block is in memory and its buffer address is on the top of the parameter stack.
DUP
Duplicate the top value on the parameter stack.
Set up for future 2+ . Duplicate the buffer address.
PREV
Place the address of the system variable PREV onto the top of the parameter stack.
Set up for ! . PREV contains the address of the most recently referenced buffer.
!
Store the specified value on the top of the parameter stack into the specified memory location.
Set PREV to point to the current buffer. If a disk access was performed, USE will now equal PREV .
THEN
BLOC1
This is the entry point from the first IF in BLOCK (i.e., The IF just prior to the BEGIN).
Control is passed here if the desired block is already in the most recently referenced buffer; therefore skipping the buffer scan and disk access.
R>
Remove the top of the return stack and place it onto the top of the parameter stack.
Retrieve the desired block number from the return stack.
DROP
Drop the top value from the parameter stack.
Discard it.
2+
Increment the top parameter stack value by 2.
Increment the buffer address by 2 to aim at the data portion of the buffer.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

BRANCH (–)

This execution time (Sequence 3) code is compiled into a definition (during Sequence 2) by ELSE , AGAIN , LOOP , +LOOP , and REPEAT to cause an unconditional branch.

NOTE: At entry IP will be aiming at an offset which is added to IP to cause either a forward or backward branch.

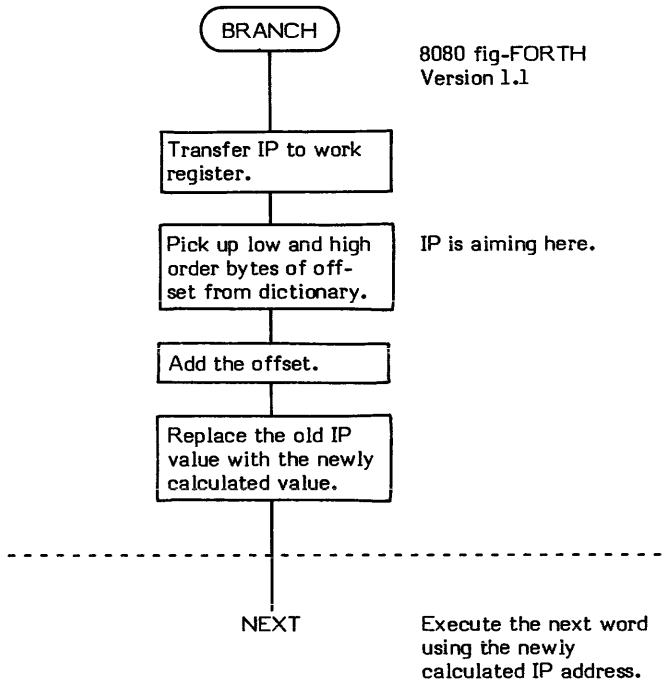
Also note that negative branch address can be obtained by using a "two's complement" value of the offset.

* **At entry** - No parameters.
* **At exit** - No parameters.

BRANCH is a low level code primitive.

Refer to ELSE , AGAIN , LOOP , +LOOP , and REPEAT .

FORTH-79: There is no FORTH-79 equivalent for BRANCH .



BUFFER

BUFFER (block number -- data address)

BUFFER is a buffer management word used to obtain the address of a buffer which may then be used for transfer of data from mass storage (usually disk) to memory. BUFFER also writes the data portion of any buffers flagged as "updated" to disk.

There are two forms of buffer management in FORTH: buffer-referencing and buffer-allocation/disk access.

The buffer-referencing management is primarily the responsibility of BLOCK (see BLOCK). Buffer-referencing management keeps track of which buffer was most recently referenced on the supposition that it is likely that this buffer will be referenced again. If the most-recently-referenced buffer does not contain the desired block number/data, the remaining buffers are also examined. If it is determined that the data is not already in memory, the data must be read from disk. Before any data transfer can occur, a buffer must be allocated. It is the purpose of BUFFER to perform and manage this allocation process.

The way in which buffers are allocated is closely related to the way buffers are logically arrayed in memory. A FORTH system normally contains several physically contiguous buffers wtih the last buffer "linked" via +BUF back to the first buffer to form a "logically circular" array of buffers.

A buffer address is obtained via the word +BUF . (See +BUF for a more detailed explanation of the buffer structure.) Given a buffer address, +BUF always returns the address of the next logical buffer in the circular queue.

Buffer allocation is performed on the basis of the "oldest allocated" buffer being re-allocated for use as the "newest allocated" buffer. (The only exception to this method is when the most recently referenced buffer, pointed to be PREV , would be overlayed. Since it is likely that the data in this buffer will be referenced again, it is wise not to allocate and overlay the contents of this buffer.)

Since BUFFER uses +BUF to obtain the address of the next buffer to allocate, buffer-allocation is also "circular". This "circular" allocation automatically causes the "oldest allocated" buffer to be used for re-allocation.

BUFFER stores the address of the next buffer to allocate (i.e., "use") into the system variable USE . Note that this is the next buffer to allocate, not the one just allocated.

Another function of buffer is to write data which has been flagged as "updated" or changed (see UPDATE) to disk. This prevents data which has been modified and flagged as updated from being overwritten with new data. This contributes to the virtual memory action of BLOCK by making explicit writes to disk of modified data unnecessary.

It is extremely important to note, however, that the simple act of modifying data in a buffer will not cause it to be written back to disk. The update flag must be set in order for this to occur.

It is also equally important to note that setting the update flag does not absolutely quarantee that data will be written to disk. This write only occurs when a buffer is allocated. If the system is restarted, or powered off, or the desired disk is removed from the drive; the data will not be written to the desired location. This problem is easily solved, though, by using the word FLUSH to force all updated buffers to be written to disk before allowing any of the conditions mentioned above to occur.

* **At entry** - The top of the parameter stack contains a 16-bit unsigned block number which will be assigned to the newly allocated buffer.

* **At exit** - The top of the parameter stack contains the 16-bit address of the beginning address of the data portion of the newly allocated buffer.

BUFFER is a high level colon definition.

Refer to BLOCK , +BUF , USE , PREV , and FLUSH .

FORTH-79: The FORTH-79 equivalent for BUFFER is BUFFER .

Definition:

```
: BUFFER  ( block number -- address )
    USE @   DUP >R
    BEGIN   +BUF   UNTIL
    USE !   R @ 0K   IF   R 2+   R @   7FFF AND   0 R/W   THEN
    R !   R PREV !   R> 2+   ;
```

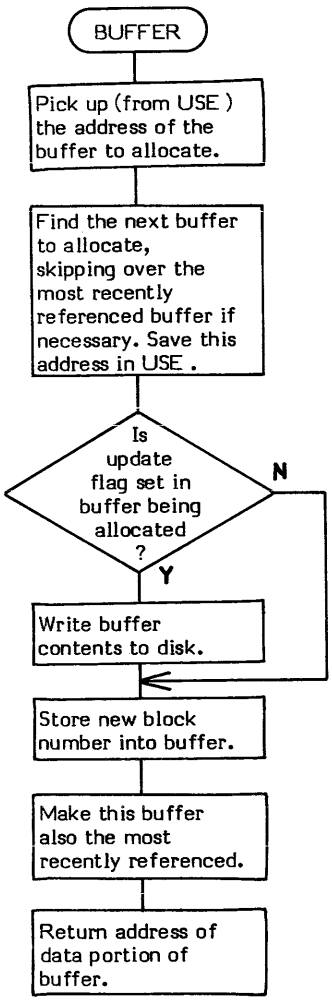
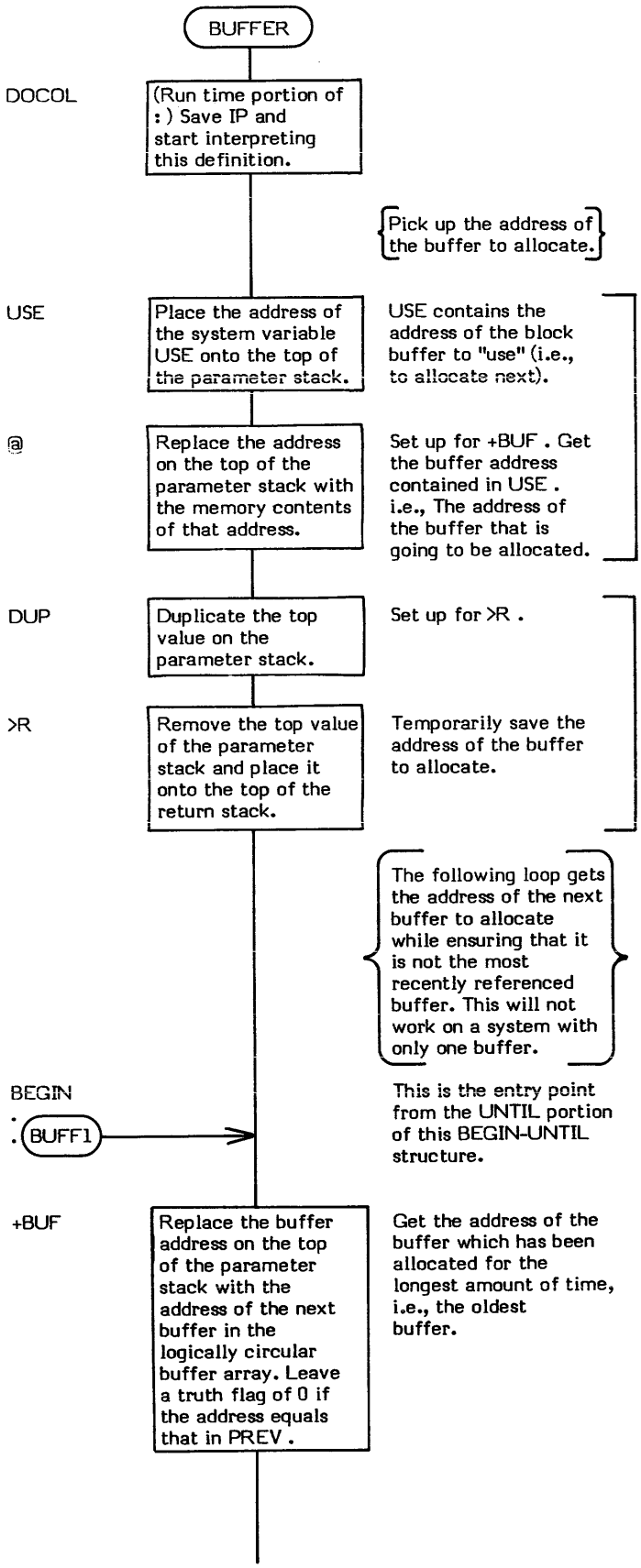
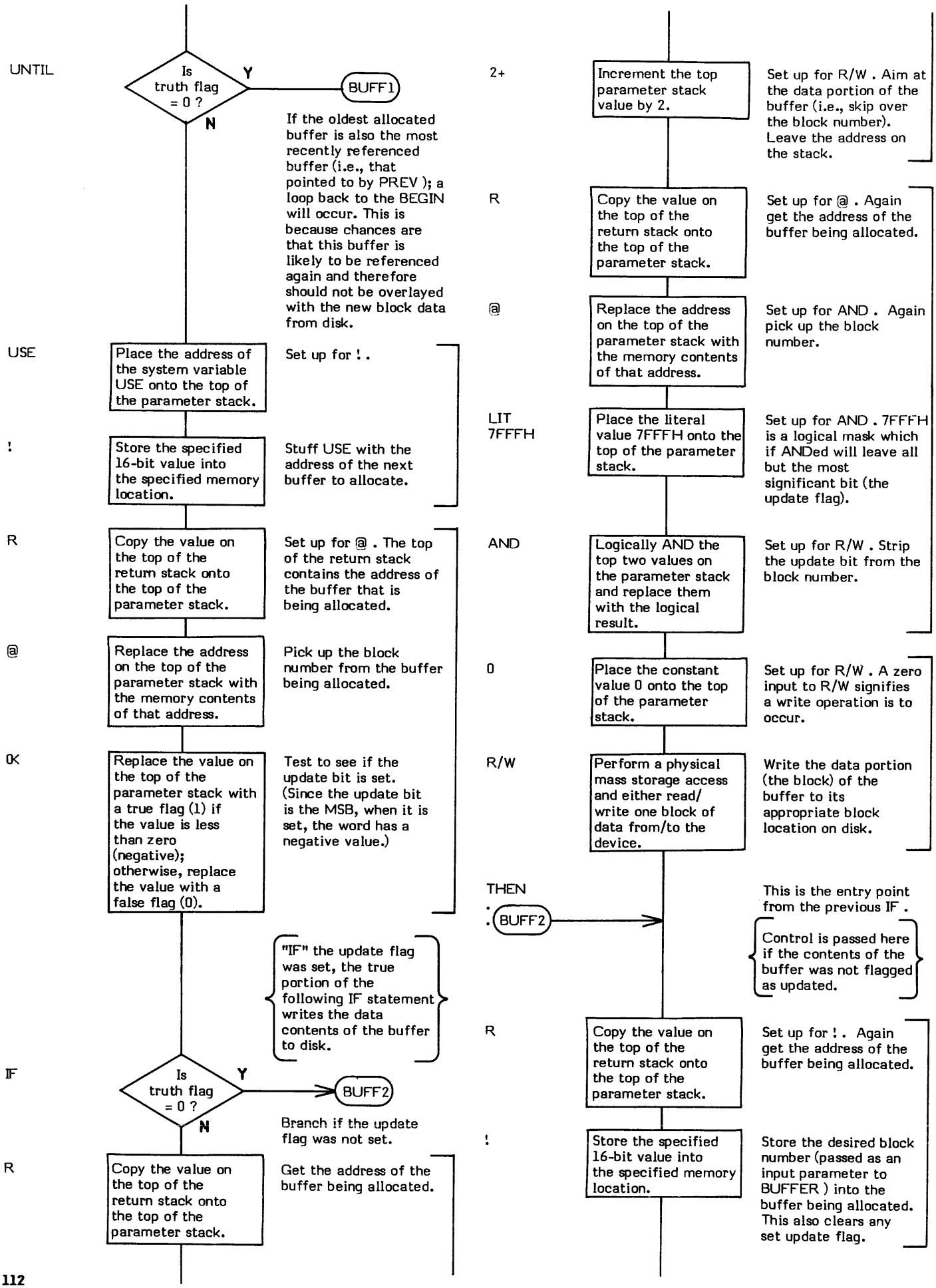
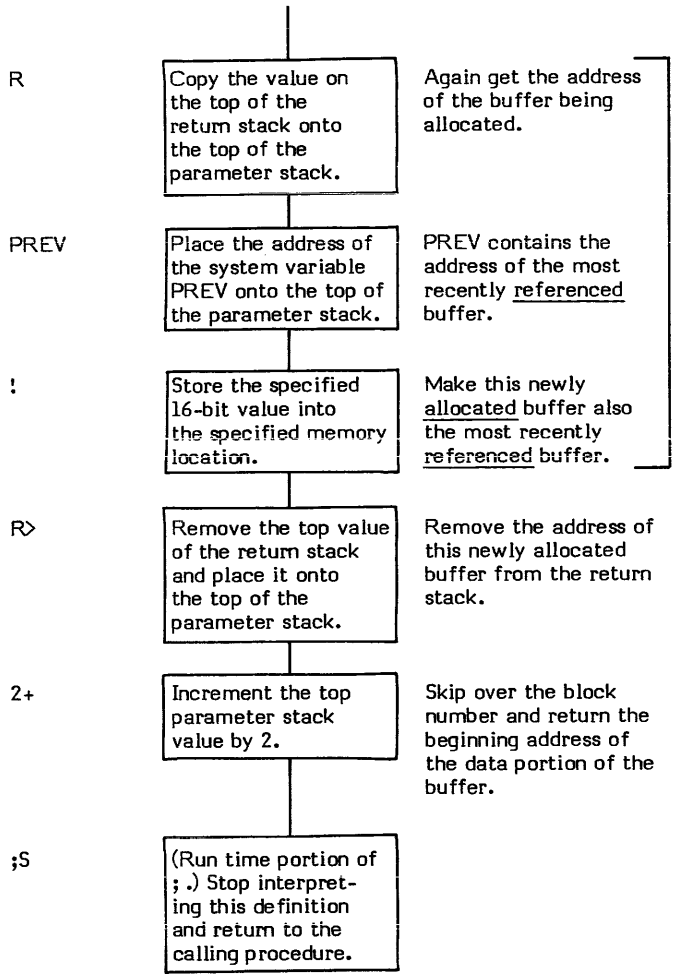


Figure BUFFER-1

High Level Flowchart of BUFFER




UNTIL
Is truth flag = 0 ?
Y
N
BUFF1
If the oldest allocated buffer is also the most recently referenced buffer (i.e., that pointed to by PREV); a loop back to the BEGIN will occur. This is because chances are that this buffer is likely to be referenced again and therefore should not be overlayed with the new block data from disk.
USE
Place the address of the system variable USE onto the top of the parameter stack.
Set up for ! .
!
Store the specified 16-bit value into the specified memory location.
Stuff USE with the address of the next buffer to allocate.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for @ . The top of the return stack contains the address of the buffer that is being allocated.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the block number from the buffer being allocated.
0K
Replace the value on the top of the parameter stack with a true flag (1) if the value is less than zero (negative); otherwise, replace the value with a false flag (0).
Test to see if the update bit is set. (Since the update bit is the MSB, when it is set, the word has a negative value.)
"IF" the update flag was set, the true portion of the following IF statement writes the data contents of the buffer to disk.
IF
Is truth flag = 0 ?
Y
N
BUFF2
Branch if the update flag was not set.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Get the address of the buffer being allocated.
2+
Increment the top parameter stack value by 2.
Set up for R/W . Aim at the data portion of the buffer (i.e., skip over the block number). Leave the address on the stack.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for @ . Again get the address of the buffer being allocated.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for AND . Again pick up the block number.
LIT 7FFFH
Place the literal value 7FFFH onto the top of the parameter stack.
Set up for AND . 7FFFH is a logical mask which if ANDed will leave all but the most significant bit (the update flag).
AND
Logically AND the top two values on the parameter stack and replace them with the logical result.
Set up for R/W . Strip the update bit from the block number.
0
Place the constant value 0 onto the top of the parameter stack.
Set up for R/W . A zero input to R/W signifies a write operation is to occur.
R/W
Perform a physical mass storage access and either read/ write one block of data from/to the device.
Write the data portion (the block) of the buffer to its appropriate block location on disk.
THEN
BUFF2
This is the entry point from the previous IF .
Control is passed here if the contents of the buffer was not flagged as updated.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for ! . Again get the address of the buffer being allocated.
!
Store the specified 16-bit value into the specified memory location.
Store the desired block number (passed as an input parameter to BUFFER) into the buffer being allocated. This also clears any set update flag.


R
Copy the value on the top of the return stack onto the top of the parameter stack.
Again get the address of the buffer being allocated.
PREV
Place the address of the system variable PREV onto the top of the parameter stack.
PREV contains the address of the most recently referenced buffer.
!
Store the specified 16-bit value into the specified memory location.
Make this newly allocated buffer also the most recently referenced buffer.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Remove the address of this newly allocated buffer from the return stack.
2+
Increment the top parameter stack value by 2.
Skip over the block number and return the beginning address of the data portion of the buffer.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

C!

C! (byte \ address —)

C! (pronounced "C-store") stores a byte (or "character"--hence "C"-store) from the top of the stack into the specified memory location.

Word addressing computers may need further specification.

! is the word used to store 16-bit values into memory. C@ has the opposite effect of C! .

HOLD is an example of a word which uses C!

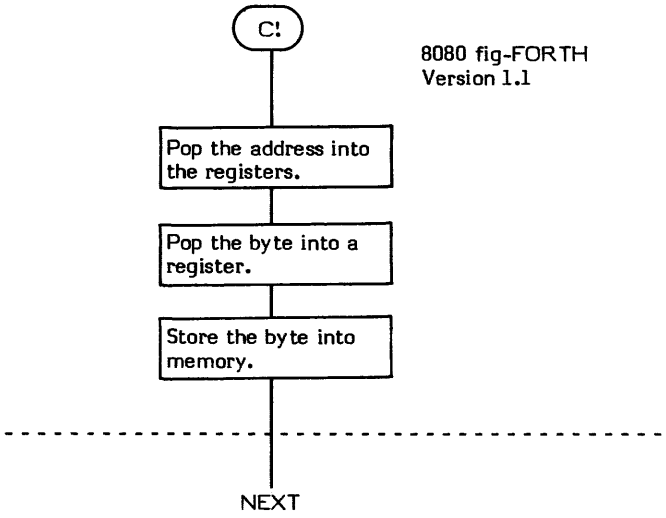
* **At entry** - The top of the parameter stack contains the 16-bit address specifying the location the byte is to be stored into. The second entry contains a 16-bit word. The low order 8-bits are stored into memory. The high order 8-bits are ignored.

* **At exit** - No parameters.

C! is a low level code primitive.

Refer to C@ , and ! .

FORTH-79: The FORTH-79 equivalent for C! is C! .



C, (single byte value —)

C, (pronounced "C-comma") stores the low order single byte, or character (hence, "C" comma) from the top of the parameter stack into the next available dictionary location and advances the dictionary pointer.

The word , is used to store 16-bit values.

* **At entry** - The low order 8-bits of the top parameter stack value contain the value to be stored into the dictionary.
* **At exit** - No parameters.

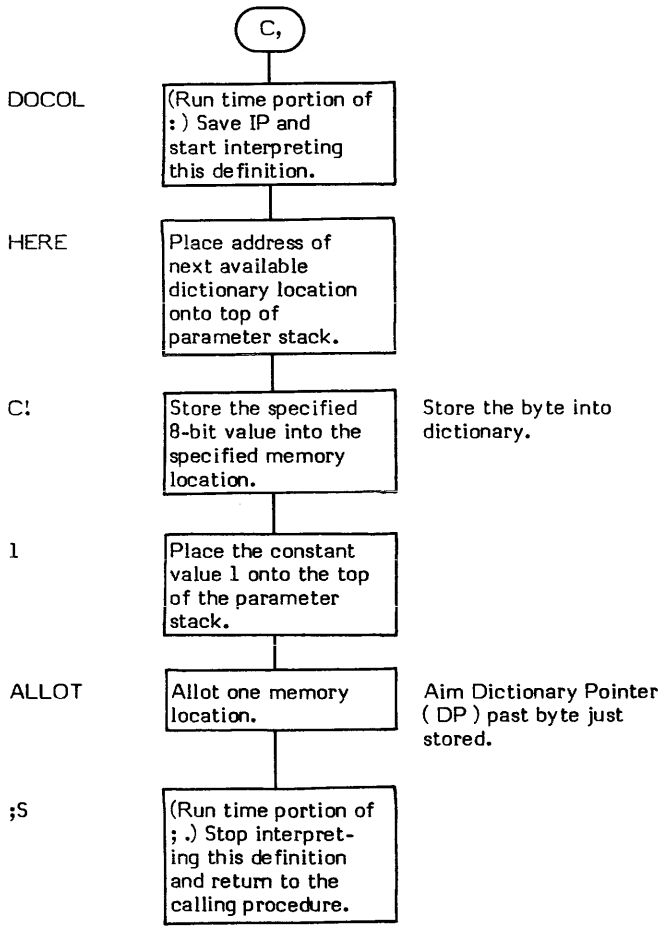
C, is a high level colon definition.

Refer to DP .

FORTH-79: C, is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:

```
:  C,  ( byte value -- )
       HERE  C!  1  ALLOT  ;
```



C/L

C/L (-- characters per editing line)

C/L (pronounced "C-slash-L") is a single precision CONSTANT value. This constant places the number of characters per editing line onto the top of the parameter stack. FORTH editing screens normally consist of 1024 (decimal) characters organized as 16 lines of 64 characters.

VLIST is an example of a word which uses C/L.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit single precision value of the number of characters per editing line.

FORTH-79: There is no FORTH-79 equivalent for C/L .

C@ (address — byte)

C@ (pronounced "C-fetch") replaces an address on the top of the parameter stack with the 8-bit contents of that memory location.

Note that @ is used to fetch 16-bit values from memory.

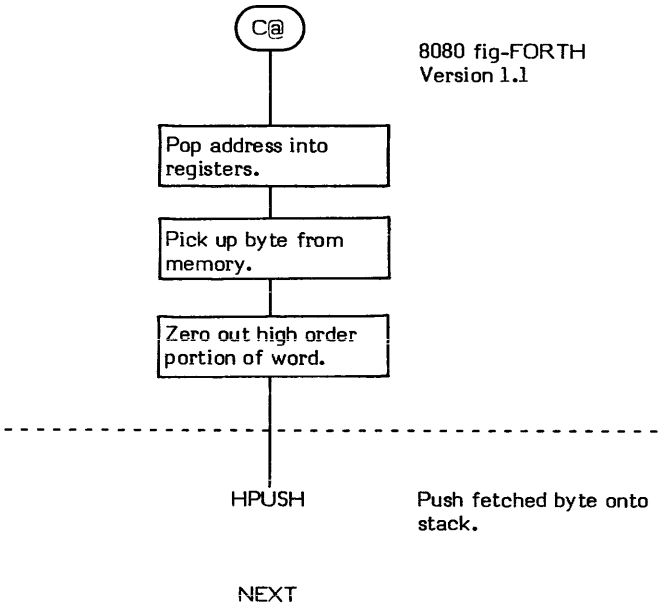
CREATE is an example of a word which uses C@.

* **At entry** - The top of the parameter stack contains the 16-bit address specifying the memory location from which the byte will be fetched.
* **At exit** - The low order 8-bits of the top of the parameter stack contain the 8-bit contents of the specified memory address. The high order 8-bits of the top of the parameter stack contain zeros.

C@ is a low level code primitive.

Refer to @ .

FORTH-79: The FORTH-79 equivalent for C@ is C@ .



CFA

CFA (Parameter Field Address -- Code Field Address)

CFA (pronounced "C-F-A") converts a given Parameter Field Address (PFA) of a dictionary definition into its Code Field Address (CFA).

The structure of the header of a FORTH definition is:

Name Field	Variable length
Link Field	2 byte address pointer
Code Field	2 byte address pointer
Parameter Field	Variable length

An example of the use of CFA can be found in the word (;CODE) .

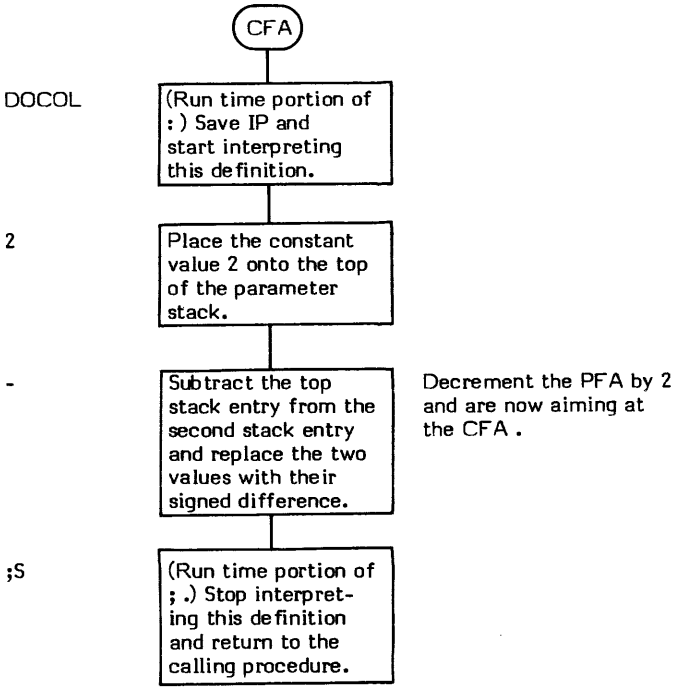
* **At entry** - The top of the parameter stack contains the 16-bit Parameter Field Address of a FORTH definition.
* **At exit** - The top of the parameter stack contains the 16-bit Code Field Address of the specified FORTH definition.

CFA is a high level colon definition.

Refer to NFA , LFA , and PFA .

FORTH-79: There is no FORTH-79 equivalent for CFA . A FORTH-79 program may not address into a definition's Code Field.

Definition:
```
:  CFA  ( PFA -- CFA )
     2  -  ;
```



CMOVE (**source address\ destination address\ length** —)

CMOVE (pronounced "C-move") is a character (byte) oriented word which moves a character string byte-by-byte from the source address to the destination proceeding towards high memory. Note that bytes are moved from the beginning of the string first; therefore the destination string cannot overlap the source string. (Some versions of FORTH use the word <CMOVE (backwards CMOVE) to move strings which overlap in this manner.)

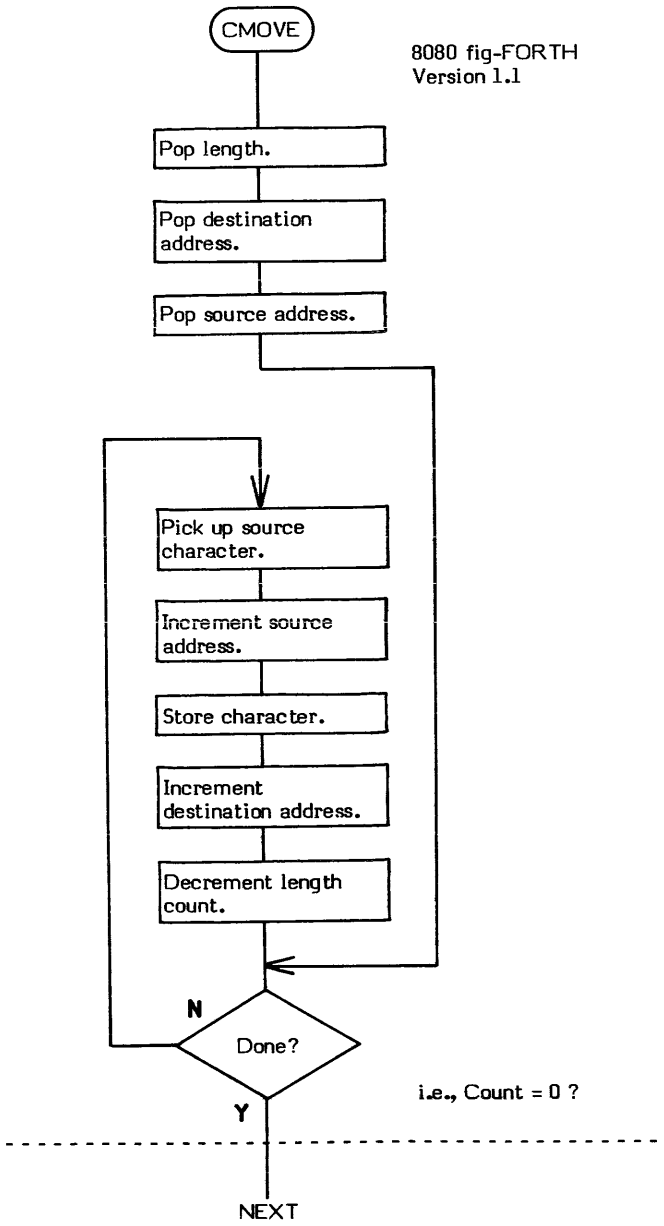
Note, however, that if the destination address is 1 byte higher than the source address, CMOVE will have the effect of rippling the first byte throughout the specified length of memory. This ripple effect only works for read/write memory. It will not work for write-only memory (such as specialized video displays). This is because the byte to be moved must be read from memory.

* **At entry** - The top of the parameter stack contains the unsigned 16-bit length of the string to move. The second entry contains the destination address and the third entry contains the source address.

* **At exit** - No parameters.

CMOVE is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for CMOVE is CMOVE .



COLD

COLD (–)

COLD is the "cold-start" routine for the FORTH system. The purpose of COLD is to initialize user variables to their startup values and then call ABORT .

COLD may be executed from the terminal. Its effect is to remove all definitions except the basic FORTH vocabulary and reset the system.

COLD performs seven basic tasks (8080 fig-FORTH version 1.1):

1. Empties the buffers.
2. Sets disk density.
3. Initializes USE and PREV buffer pointers.
4. Selects Drive 0 as the mass storage device.
5. Disables the printer.
6. Initializes the following USER variables: S0, R0, TIB, WIDTH, WARNING, FENCE, DP, and VOC-LINK.
7. Calls ABORT .

NOTE: The exact nature of COLD is installation dependent.

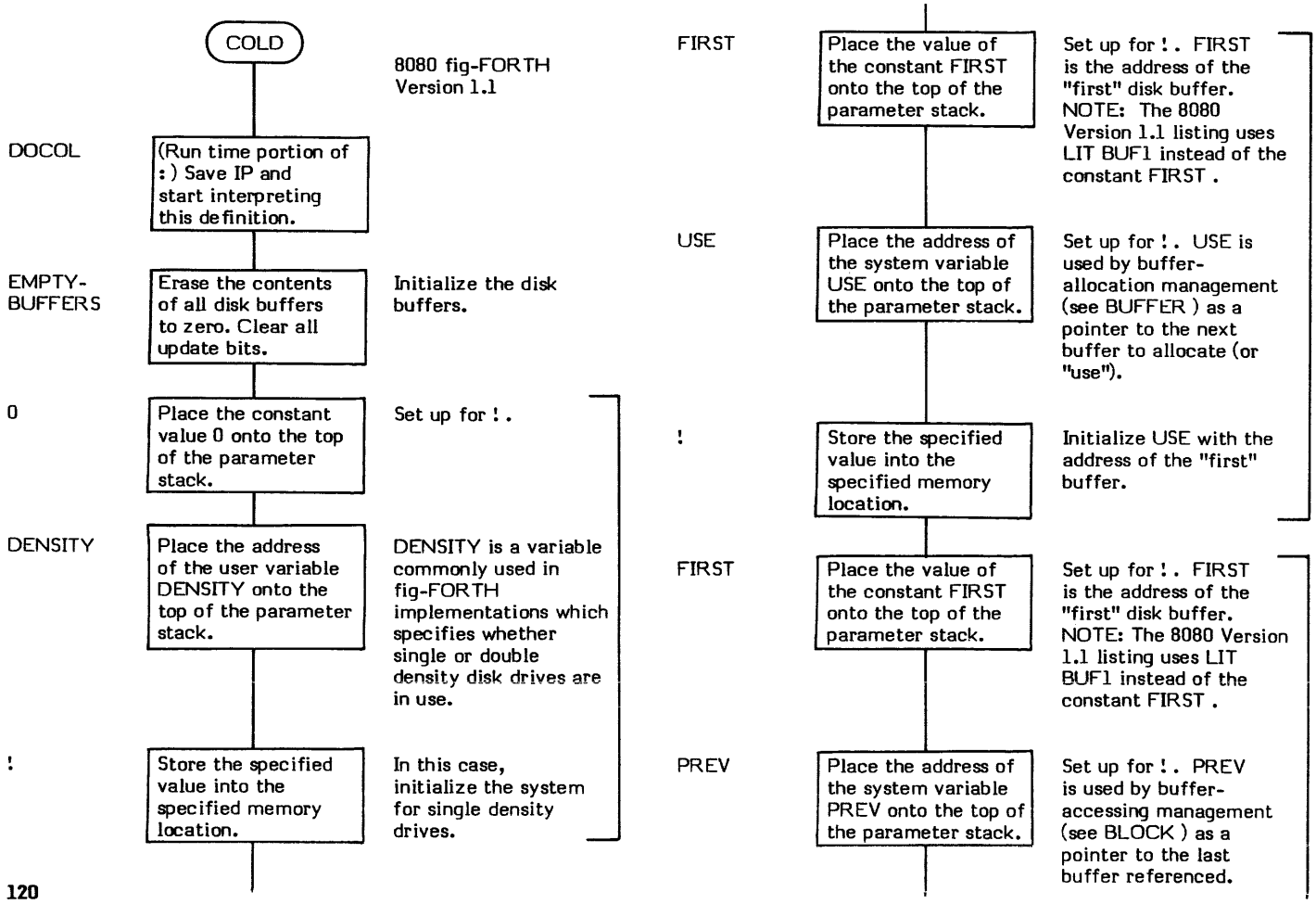
* **At entry** - No parameters.
* **At exit** - No parameters.

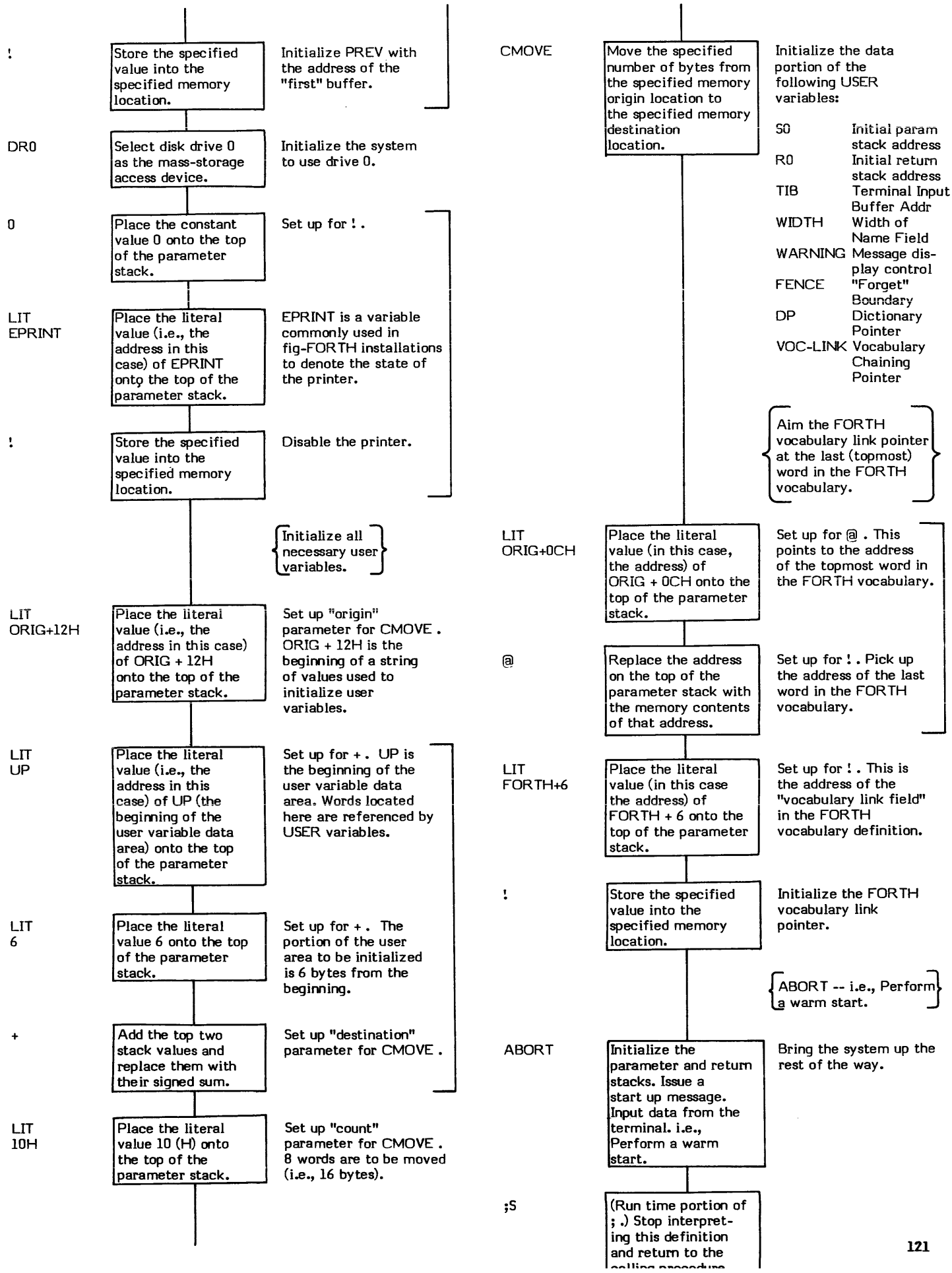
COLD is a high level colon definition.

FORTH-79: There is no FORTH-79 equivalent for COLD .

Definition:

```
: COLD ( -- ) ( 8080 Version 1.1 )
    EMPTY-BUFFERS  0 DENSITY !  FIRST USE !  FIRST PREV !  DR0
    0 EPRINT !  ORG+12  UP 6 +  10 CMOVE  ORIG+C @  FORTH+6  !
    ABORT ;
```




!
Store the specified value into the specified memory location.
Initialize PREV with the address of the "first" buffer.
DR0
Select disk drive 0 as the mass-storage access device.
Initialize the system to use drive 0.
0
Place the constant value 0 onto the top of the parameter stack.
Set up for ! .
LIT
EPRINT
Place the literal value (i.e., the address in this case) of EPRINT onto the top of the parameter stack.
EPRINT is a variable commonly used in fig-FORTH installations to denote the state of the printer.
!
Store the specified value into the specified memory location.
Disable the printer.
Initialize all necessary user variables.
LIT
ORIG+12H
Place the literal value (i.e., the address in this case) of ORIG + 12H onto the top of the parameter stack.
Set up "origin" parameter for CMOVE . ORIG + 12H is the beginning of a string of values used to initialize user variables.
LIT
UP
Place the literal value (i.e., the address in this case) of UP (the beginning of the user variable data area) onto the top of the parameter stack.
Set up for + . UP is the beginning of the user variable data area. Words located here are referenced by USER variables.
LIT
6
Place the literal value 6 onto the top of the parameter stack.
Set up for + . The portion of the user area to be initialized is 6 bytes from the beginning.
+
Add the top two stack values and replace them with their signed sum.
Set up "destination" parameter for CMOVE .
LIT
10H
Place the literal value 10 (H) onto the top of the parameter stack.
Set up "count" parameter for CMOVE . 8 words are to be moved (i.e., 16 bytes).
CMOVE
Move the specified number of bytes from the specified memory origin location to the specified memory destination location.
Initialize the data portion of the following USER variables:
S0 Initial param stack address
R0 Initial return stack address
TIB Terminal Input Buffer Addr
WIDTH Width of Name Field
WARNING Message display control
FENCE "Forget" Boundary
DP Dictionary Pointer
VOC-LINK Vocabulary Chaining Pointer
Aim the FORTH vocabulary link pointer at the last (topmost) word in the FORTH vocabulary.
LIT
ORIG+0CH
Place the literal value (in this case, the address) of ORIG + 0CH onto the top of the parameter stack.
Set up for @ . This points to the address of the topmost word in the FORTH vocabulary.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for ! . Pick up the address of the last word in the FORTH vocabulary.
LIT
FORTH+6
Place the literal value (in this case the address) of FORTH + 6 onto the top of the parameter stack.
Set up for ! . This is the address of the "vocabulary link field" in the FORTH vocabulary definition.
!
Store the specified value into the specified memory location.
Initialize the FORTH vocabulary link pointer.
ABORT -- i.e., Perform a warm start.
ABORT
Initialize the parameter and return stacks. Issue a start up message. Input data from the terminal. i.e., Perform a warm start.
Bring the system up the rest of the way.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the

COMPILE

COMPILE

COMPILE TIME: (--)
(Sequence 2)

EXECUTION TIME: (--)
(Sequence 3)

COMPILE is a "form" of a compiler word and therefore has two sets of actions; an apparent compile time action (Sequence 2) and an execution time action (Sequence 3).

COMPILE causes the CFA of the word immediately following COMPILE (in the input data stream) to be compiled into a definition. The CFA to be compiled is specified at Sequence 1 time when COMPILE is itself compiled into a compiler or defining word (i.e., the "parent").

COMPILE TIME -- How this "CFA specification" occurs is quite simple. When COMPILE is encountered in the input data stream, its CFA is automatically compiled by the Interpreter. When the word following CFA is encountered, (i.e., the "specified" word) its CFA is also automatically compiled into the dictionary immediately following COMPILE 's CFA . Note that this is a normal compilation sequence. There is no correlation between COMPILE and the word "to be compiled" except that the CFA of the word "to be compiled" immediately follows the CFA of COMPILE .

EXECUTION TIME -- Since COMPILE performs compiling, it is important to keep in mind that COMPILE 's execution time (Sequence 3) occurs at the word "to be compiled's" (i.e., the "child's") compilation time (Sequence 2).

The specified CFA is actually compiled into a definition at Sequence 2 time when the compiler/defining (the "parent") word executes (creates a "child") thereby executing COMPILE .

When COMPILE executes, the CFA it is to compile into the definition being compiled is physically located in the next Parameter Field location following COMPILE 's CFA . This is also the location of the next CFA to be executed and therefore the address of this location has been placed onto the top of the return stack by DOCOL .

COMPILE obtains this address from the return stack, uses it to fetch the specified CFA, then increments the address by 2 and returns it to the return stack. This skips over the "specified" CFA and will cause the location following the specified CFA to be executed when COMPILE finishes.

An example of the use of COMPILE is the conditional branch word IF . The execution time procedure for IF is 0BRANCH . This is specified within the source code for IF as COMPILE 0BRANCH . At Sequence 1 when IF is compiled, the CFA 's for COMPILE and 0BRANCH are compiled into IF 's definition. Later, during Sequence 2, when IF is executed, the CFA for 0BRANCH is compiled into the "child's" Parameter Field.

This concept of specifing the name of the definition to be compiled at Sequence 1 but not actually compiling it into a definition until Sequence 2 is called "deferred compilation".

COMPILE TIME (Sequence 2):

* **At entry** - No parameters however COMPILE must be followed by the word desired to be compiled into the dicitionary.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.

* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

? pronounced "HUH?" (0) -- The word in question cannot be found in the dictionary.

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

COMPILE is a high level colon definition.

Refer to INTERPRET , and IF .

FORTH-79: The FORTH-79 equivalent for COMPILE is COMPILE .

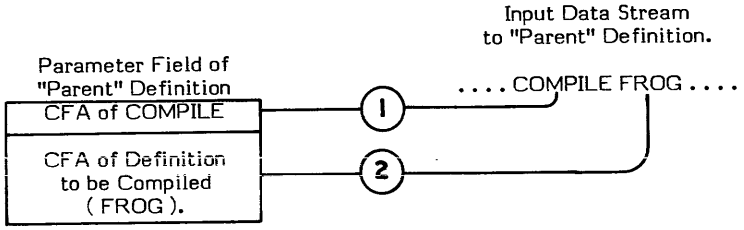
Definition:

```
:  COMPILE  ( -- )  ( execution time )
     /COMP  >R DUP 2+ R>  @ ,  ;
```

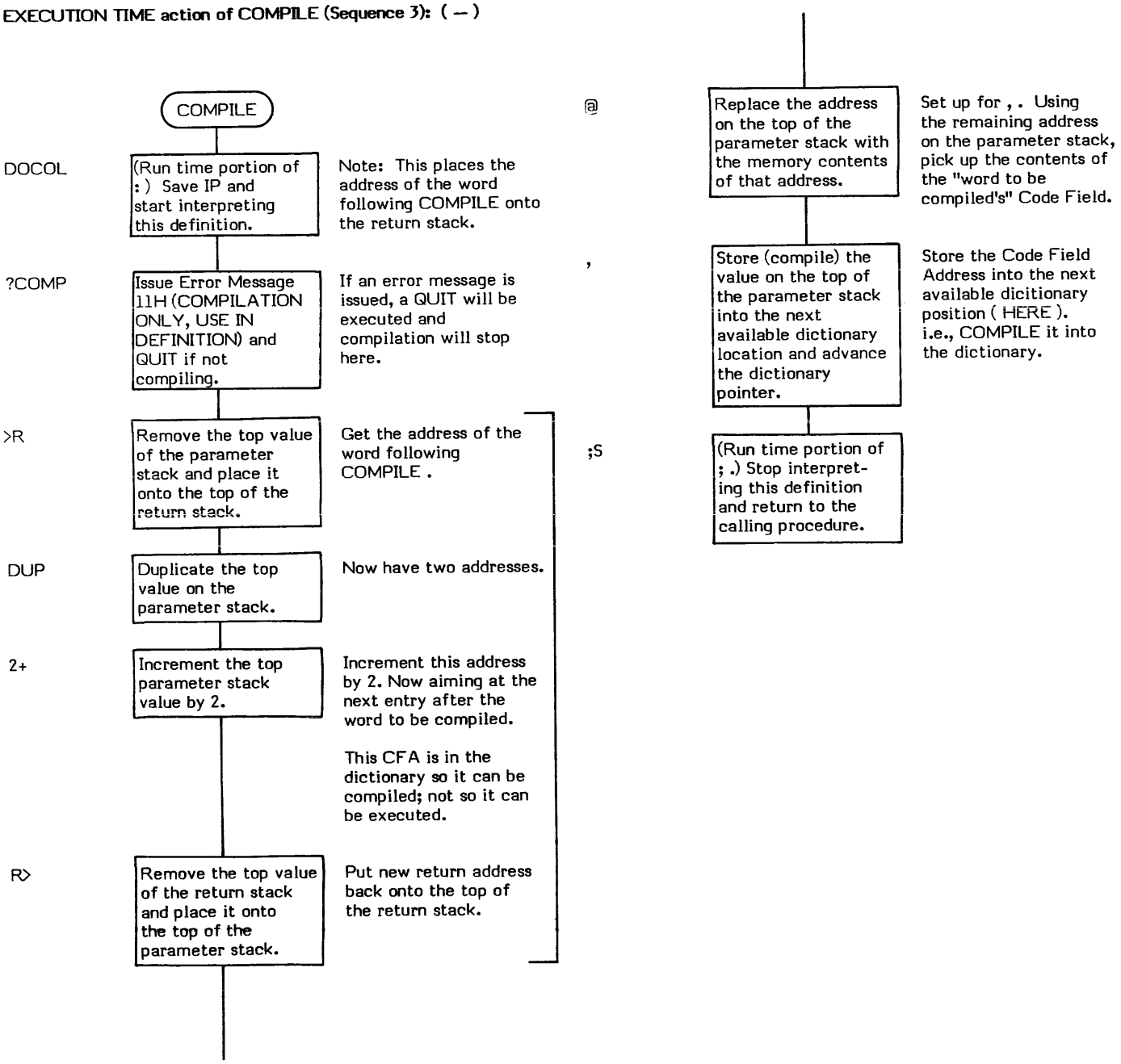
COMPILE TIME action of COMPILE (Sequence 2): (—)

NOTE: Although COMPILE performs compilation, it has no compile time action itself. Instead, COMPILE relies upon the normal compiler mechanism of the Interpreter to do the following:

1. The Interpreter must compile the CFA of COMPILE into the parent definition.
2. The Interpreter must compile the CFA of the definition whose CFA is to be eventually compiled into the parent definition.



EXECUTION TIME action of COMPILE (Sequence 3): (—)



CONSTANT

CONSTANT

COMPILE TIME): (value —)
(Sequence 2)

EXECUTION TIME: (— value)
(Sequence 3)

CONSTANT is a defining word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

A CONSTANT in FORTH has the same effect as a constant in most other computer languages. That is, a memory location containing a fixed value that can be referenced via a name. CONSTANT does differ from other languages in that it is active. When the constant name is executed, the constant value is placed onto the top of the parameter stack.

The compile time action of CONSTANT is to define a named 16-bit constant in the form:

n CONSTANT cccc

where n is the constant's value and cccc is the value's assigned name (e.g., 26 CONSTANT MAX-LETTERS).

The execution time action of CONSTANT , when the named constant is referenced, is to push this 16-bit value onto the top of the parameter stack.

It is possible to change the value of a RAM-based constant but this is extremely poor programming practice. VARIABLE should be used if the value must change.

An example of the use of CONSTANT are the words LIMIT and FIRST . These words are used in +BUF .

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit single precision constant value. The constant's name must immediately follow CONSTANT in the input stream.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the previously defined 16-bit value.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) -- The dictionary has grown into the Terminal Input Buffer.

DEFINITION NOT FINISHED (14H) — The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

CONSTANT is a high level colon definition.

Refer to VARIABLE , and +BUF .

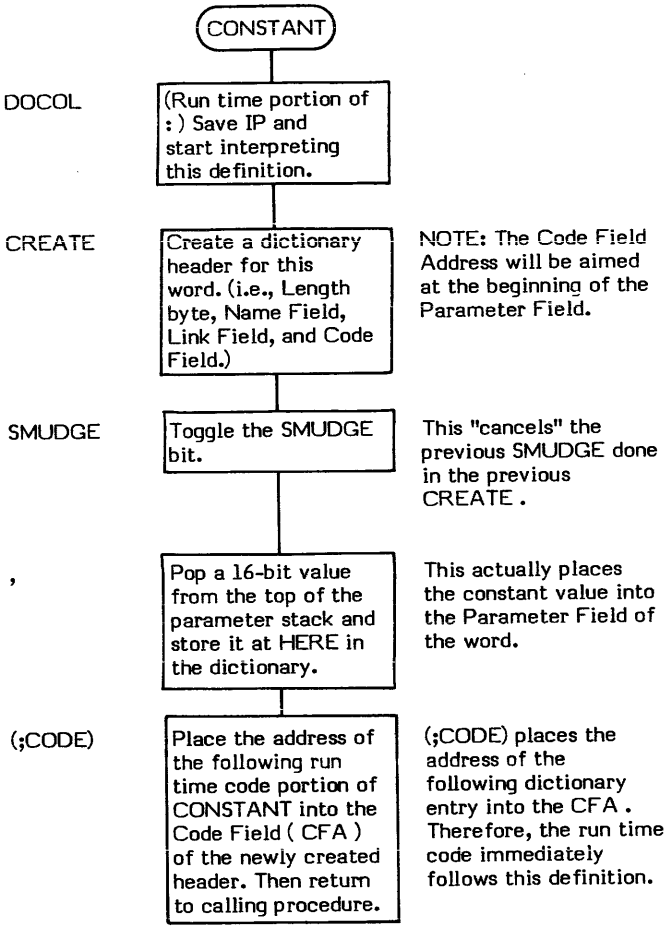
FORTH-79: The FORTH-79 equivalent for CONSTANT is CONSTANT .

Definition:

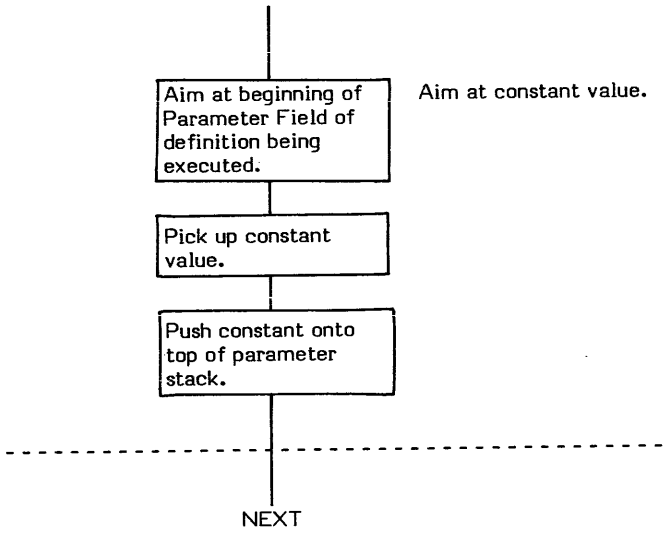
```
:  CONSTANT   ( value -- )   ( compile time )
       CREATE   SMUDGE   ,   ;CODE
```

Note: The assembly language execution time code follows the ;CODE .

COMPILE TIME action of CONSTANT: (value --)
(Sequence 2)



EXECUTION TIME action of CONSTANT : (-- value)
(Sequence 3)



NOTE: The execution time (Sequence 3) code for CONSTANT physically and immediately follows the ;CODE in the 8080 fig-FORTH Version 1.1.

CONTEXT

CONTEXT (— data address)

CONTEXT is a user variable which contains a pointer to the vocabulary which is to be searched first.

CONTEXT is set to point to a specific vocabulary by executing that vocabulary name. For example, FORTH causes CONTEXT to be aimed at the FORTH vocabulary. At system startup, CONTEXT is initialized to FORTH by ABORT which is called by COLD . Note that this means that any time an ABORT occurs CONTEXT will be aimed at the FORTH vocabulary.

The use of CONTEXT is fully explained in the description of the word VOCABULARY .

The user variable CONTEXT is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

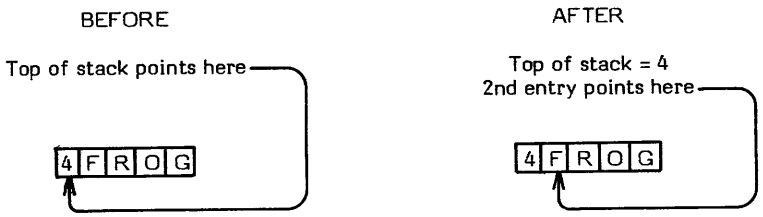
* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable CONTEXT .

Refer to VOCABULARY , ABORT , and USER .

FORTH-79: The FORTH-79 equivalent for CONTEXT is CONTEXT .

COUNT (**text string address -- text address \ char count**)

COUNT replaces the address of a text string with the byte length and byte address of the text string. The first byte of the text string must contain the string length. The text must begin at the second byte. (WORD automatically provides this format.)



COUNT is often used before TYPE .

." is an example of a word which uses COUNT.

* **At entry** - The top of the parameter stack contains the address of the beginning of a text string (the length byte). This first byte of the string must contain the length of the text (not including the length byte itself).

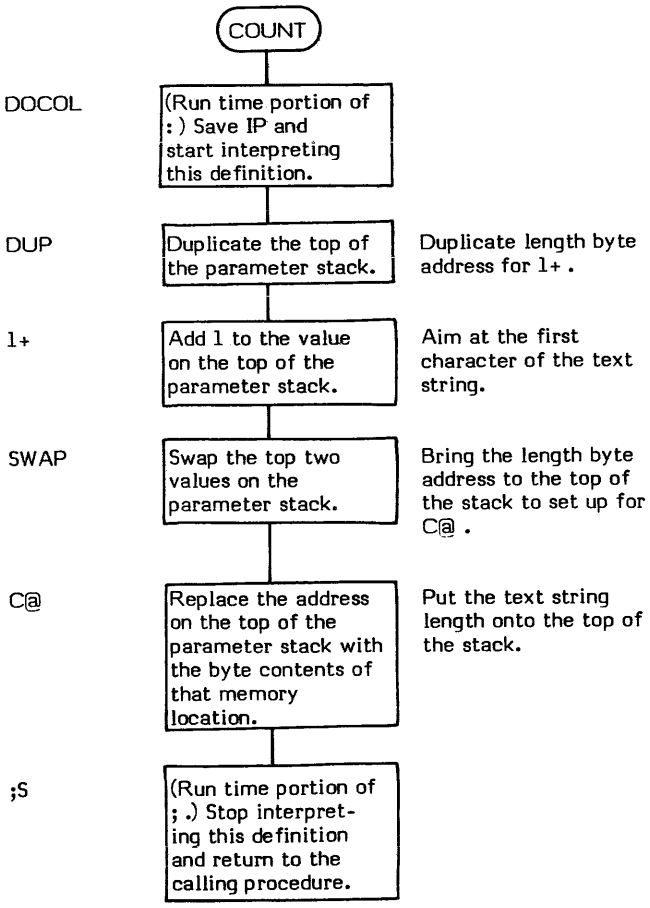
* **At exit** - The top of the parameter stack contains the length of the text string. The second entry contains the actual beginning text address.

COUNT is a high level colon definition.

Refer to TYPE .

FORTH-79: The FORTH-79 equivalent for COUNT is COUNT .

Definition:
```
:  COUNT  ( text string address -- text address \ char count )
     DUP  1+  SWAP  C@  ;
```



CR

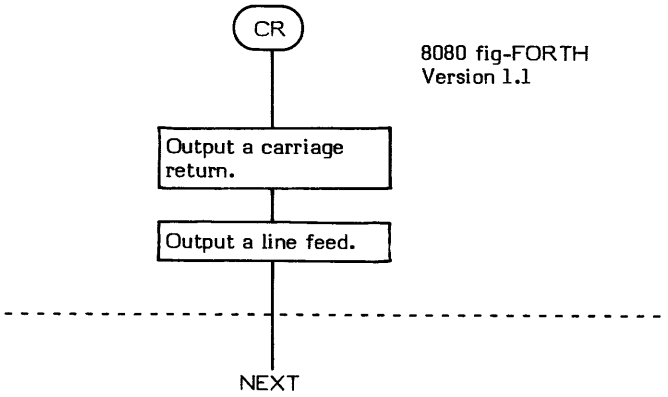
CR (–)

CR (pronounced "carriage-return") is an installation dependent word which usually transmits a carriage return and a line feed to the selected output device.

* **At entry** - No parameters.
* **At exit** - No parameters.

CR is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for CR is CR .



CREATE (–)

CREATE is a defining word used to "create" the header portion of a FORTH definition. It is used in the form:

CREATE definition name

CREATE builds a standard FORTH header. That is to say that it creates a Name Field, a Link Field, and a Code Field. Words that directly use CREATE are classified as being "defining words".

The beginning and ending bytes of the Name Field have their high order bits set. The "smudge" bit in the Name Field is also set. The maximum number of characters saved in the Name Field is controlled by the value stored in the user variable WIDTH .

Before creating a header, CREATE checks both the CONTEXT and CURRENT vocabularies to determine if the new definition name is unique. If it is not, the message "ISN'T UNIQUE" is issued. The definition header is created in either case.

The Link Field is set to point to the last definition added to the vocabulary. The "current" vocabulary's "vocabulary pseudo link field" (not to be confused with the VOC-LINK field) is changed to point to the definition being "created". (See VOCABULARY .)

The Code Field is set to point to the Parameter Field which follows the header. Note that no Parameter Field space is allocated.

: (colon) is an example of a word which uses CREATE .

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) – The dictionary has grown into the Terminal Input Buffer.

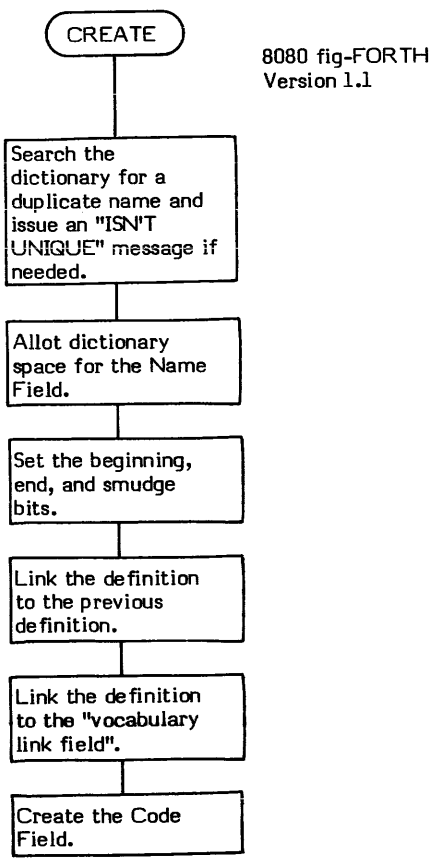
CREATE is a high level colon definition.

Refer to WIDTH , VOCABULARY , CURRENT , WARNING and : .

FORTH-79: In FORTH-79 the word CREATE now has the same meaning as the fig-FORTH <BUILDS . Refer to FORTH-79 Standard.

Definition:

```
:  CREATE  ( -- )  ( 8080 Version 1.1 )
       -FIND
       IF   DROP   NFA ID.   4 MESSAGE SPACE   THEN
       HERE   DUP   C@ WIDTH @ MIN   1+ ALLOT   DUP A0 TOGGLE
       HERE 1 -   80 TOGGLE   LATEST ,   CURRENT @ !
       HERE 2+ ,   ;
```



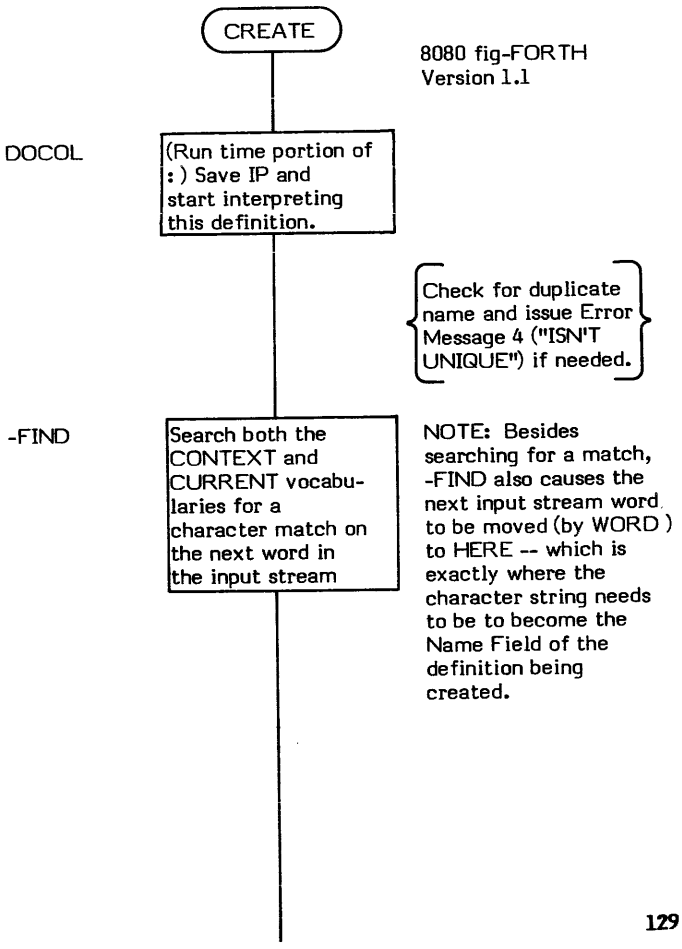
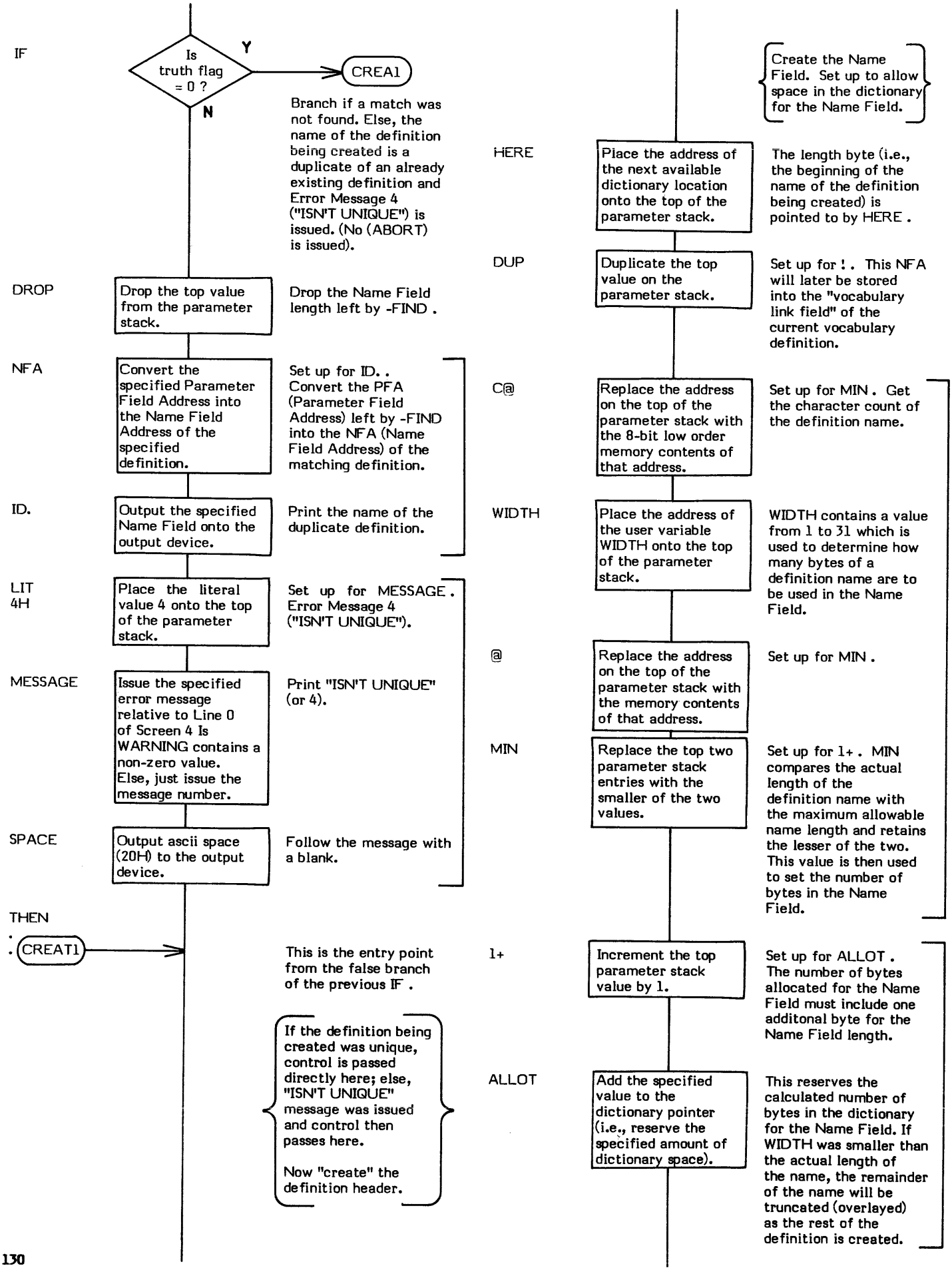
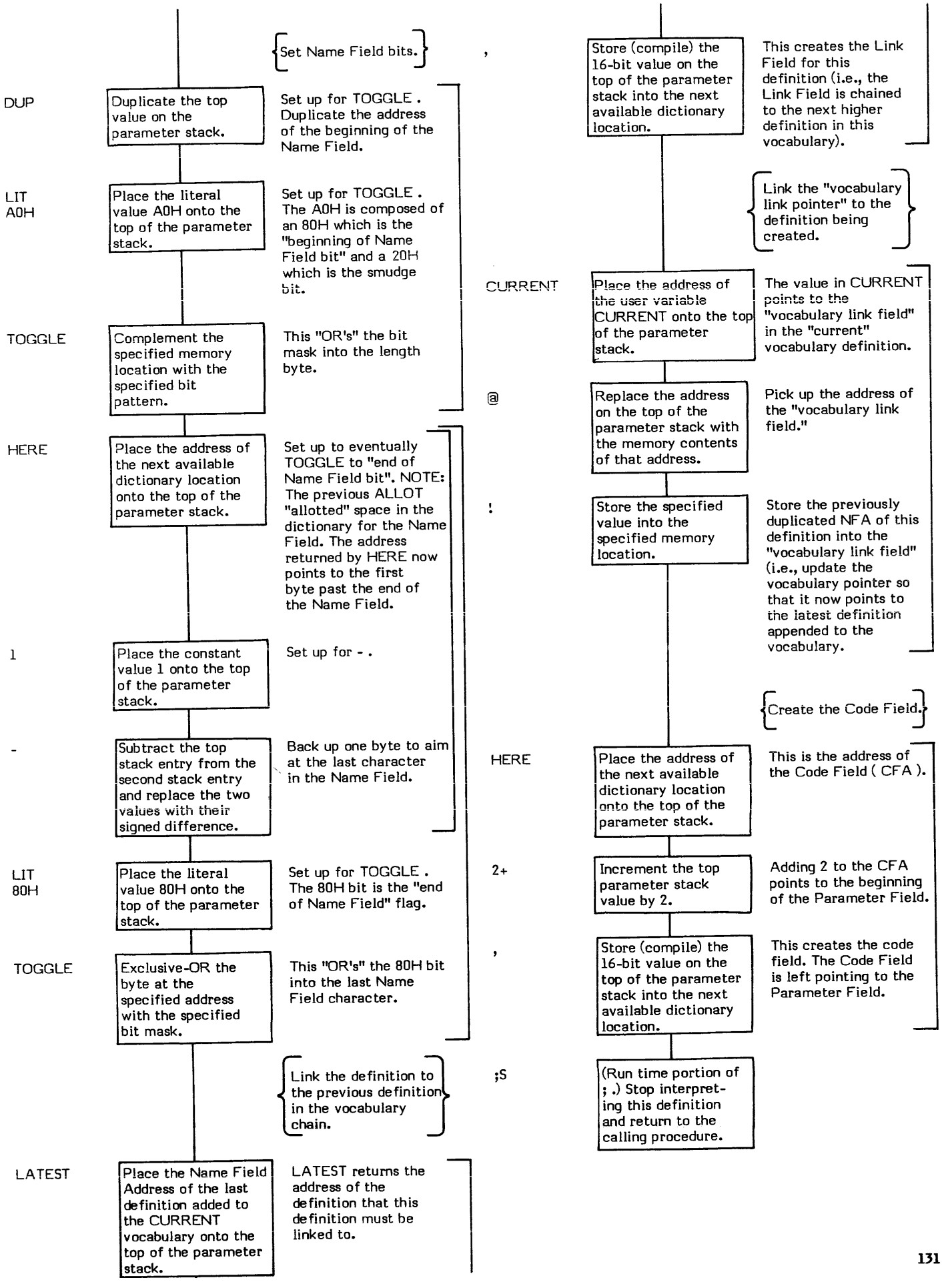


Figure CREATE-1
High Level Flow Chart of CREATE


IF
Is truth flag = 0 ?
Y
N
CREA1
Branch if a match was not found. Else, the name of the definition being created is a duplicate of an already existing definition and Error Message 4 ("ISN'T UNIQUE") is issued. (No (ABORT) is issued).
DROP
Drop the top value from the parameter stack.
Drop the Name Field length left by -FIND .
NFA
Convert the specified Parameter Field Address into the Name Field Address of the specified definition.
Set up for ID. . Convert the PFA (Parameter Field Address) left by -FIND into the NFA (Name Field Address) of the matching definition.
ID.
Output the specified Name Field onto the output device.
Print the name of the duplicate definition.
LIT
4H
Place the literal value 4 onto the top of the parameter stack.
Set up for MESSAGE . Error Message 4 ("ISN'T UNIQUE").
MESSAGE
Issue the specified error message relative to Line 0 of Screen 4 Is WARNING contains a non-zero value. Else, just issue the message number.
Print "ISN'T UNIQUE" (or 4).
SPACE
Output ascii space (20H) to the output device.
Follow the message with a blank.
THEN
CREAT1
This is the entry point from the false branch of the previous IF .
If the definition being created was unique, control is passed directly here; else, "ISN'T UNIQUE" message was issued and control then passes here.
Now "create" the definition header.
Create the Name Field. Set up to allow space in the dictionary for the Name Field.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
The length byte (i.e., the beginning of the name of the definition being created) is pointed to by HERE .
DUP
Duplicate the top value on the parameter stack.
Set up for ! . This NFA will later be stored into the "vocabulary link field" of the current vocabulary definition.
C@
Replace the address on the top of the parameter stack with the 8-bit low order memory contents of that address.
Set up for MIN . Get the character count of the definition name.
WIDTH
Place the address of the user variable WIDTH onto the top of the parameter stack.
WIDTH contains a value from 1 to 31 which is used to determine how many bytes of a definition name are to be used in the Name Field.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for MIN .
MIN
Replace the top two parameter stack entries with the smaller of the two values.
Set up for 1+ . MIN compares the actual length of the definition name with the maximum allowable name length and retains the lesser of the two. This value is then used to set the number of bytes in the Name Field.
1+
Increment the top parameter stack value by 1.
Set up for ALLOT . The number of bytes allocated for the Name Field must include one additonal byte for the Name Field length.
ALLOT
Add the specified value to the dictionary pointer (i.e., reserve the specified amount of dictionary space).
This reserves the calculated number of bytes in the dictionary for the Name Field. If WIDTH was smaller than the actual length of the name, the remainder of the name will be truncated (overlayed) as the rest of the definition is created.


Set Name Field bits.
,
DUP
Duplicate the top value on the parameter stack.
Set up for TOGGLE . Duplicate the address of the beginning of the Name Field.
LIT A0H
Place the literal value A0H onto the top of the parameter stack.
Set up for TOGGLE . The A0H is composed of an 80H which is the "beginning of Name Field bit" and a 20H which is the smudge bit.
TOGGLE
Complement the specified memory location with the specified bit pattern.
This "OR's" the bit mask into the length byte.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
Set up to eventually TOGGLE to "end of Name Field bit". NOTE: The previous ALLOT "allotted" space in the dictionary for the Name Field. The address returned by HERE now points to the first byte past the end of the Name Field.
1
Place the constant value 1 onto the top of the parameter stack.
Set up for - .
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Back up one byte to aim at the last character in the Name Field.
LIT 80H
Place the literal value 80H onto the top of the parameter stack.
Set up for TOGGLE . The 80H bit is the "end of Name Field" flag.
TOGGLE
Exclusive-OR the byte at the specified address with the specified bit mask.
This "OR's" the 80H bit into the last Name Field character.
Link the definition to the previous definition in the vocabulary chain.
;S
LATEST
Place the Name Field Address of the last definition added to the CURRENT vocabulary onto the top of the parameter stack.
LATEST returns the address of the definition that this definition must be linked to.
Store (compile) the 16-bit value on the top of the parameter stack into the next available dictionary location.
This creates the Link Field for this definition (i.e., the Link Field is chained to the next higher definition in this vocabulary).
Link the "vocabulary link pointer" to the definition being created.
CURRENT
Place the address of the user variable CURRENT onto the top of the parameter stack.
The value in CURRENT points to the "vocabulary link field" in the "current" vocabulary definition.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the address of the "vocabulary link field."
!
Store the specified value into the specified memory location.
Store the previously duplicated NFA of this definition into the "vocabulary link field" (i.e., update the vocabulary pointer so that it now points to the latest definition appended to the vocabulary.
Create the Code Field.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
This is the address of the Code Field (CFA).
2+
Increment the top parameter stack value by 2.
Adding 2 to the CFA points to the beginning of the Parameter Field.
,
Store (compile) the 16-bit value on the top of the parameter stack into the next available dictionary location.
This creates the code field. The Code Field is left pointing to the Parameter Field.
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

CSP

CSP (— data address)

CSP (pronounced "C-S-P" for Compiler Stack Pointer) is a user variable which is used as a temporary storage location for the stack pointer position.

This is normally used as a compiler security check. A word such as : will execute !CSP which stores the stack pointer position into CSP , before compiling a definition. Then, before the new definition is "un-smudged", ?CSP is executed. ?CSP compares the value stored in CSP with the current stack pointer and issues an error message if the two are not equal.

The user variable CSP is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** -The top of the parameter stack contains the address of the user variable CSP .

Refer to : , !CSP , ?CSP , and USER .

FORTH-79: There is no FORTH-79 equivalent for CSP .

CURRENT (— data address)

CURRENT is a user variable that contains a pointer to the vocabulary to which definitions are "currently" being appended to.

The use of CURRENT is fully explained in the description of the word VOCABULARY .

The user variable CURRENT is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

CURRENT is set to point to a specific vocabulary by executing the word DEFINITIONS , that copies the vocabulary pointer in CONTEXT into CURRENT . At system startup, CURRENT is initialized to FORTH by ABORT which is called by COLD . Note that this means any time an ABORT occurs, CURRENT will be aimed at the FORTH vocabulary.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable CURRENT .

Refer to DEFINITIONS , VOCABULARY , ABORT , and USER .

FORTH-79: The FORTH-79 equivalent for CURRENT is CURRENT .

D+

D+ (double \ double -- 32-bit sum)

D+ (pronounced "D-plus") adds the top two 32-bit signed values on the parameter stack and replaces them with their 32-bit signed sum.

Note that generation of a carry goes unnoticed.

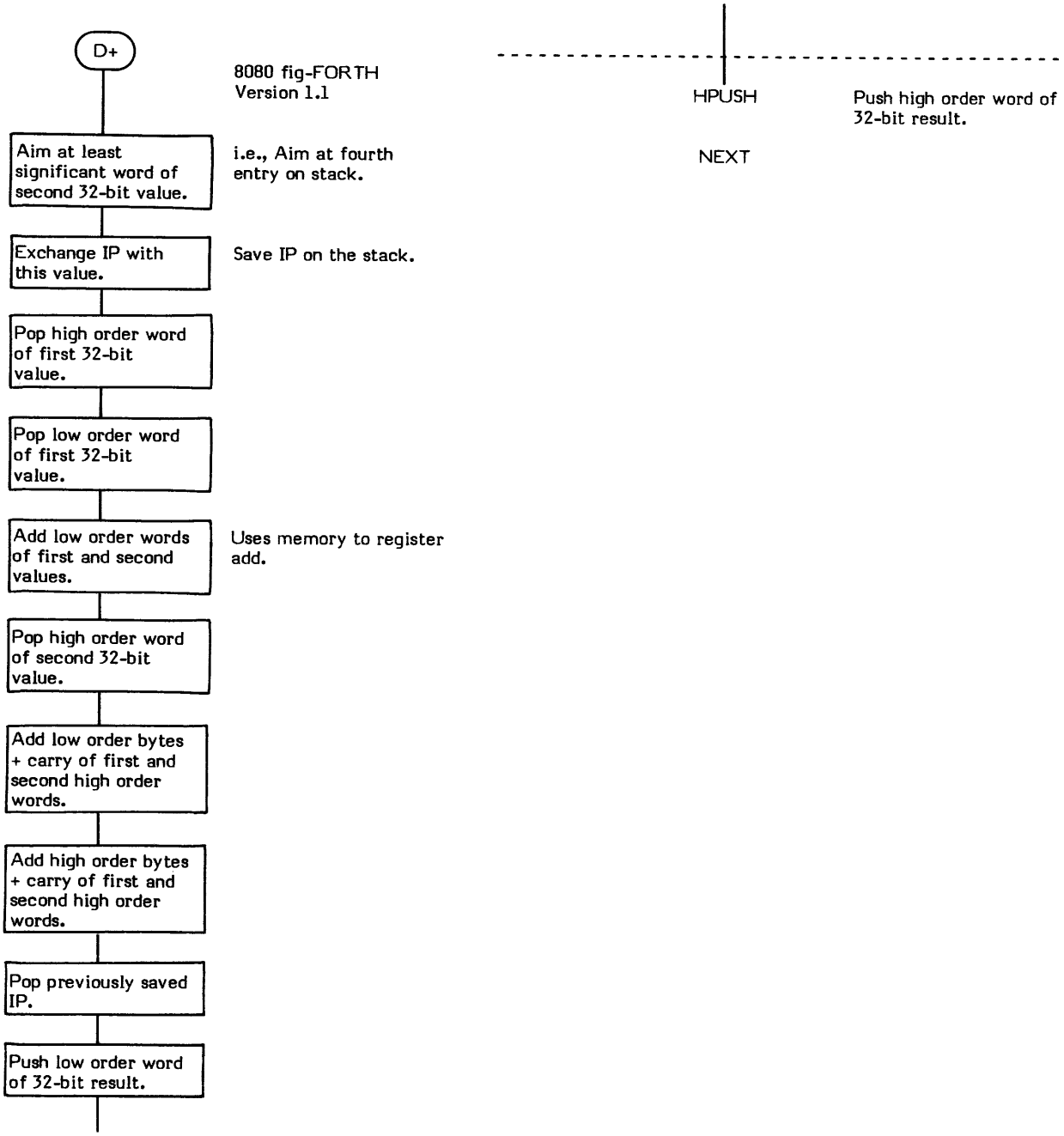
(NUMBER) is an example of a word which uses D+ .

* **At entry** - The top and second words of the parameter stack contain a 32-bit signed double precision value with the signed, most significant portion on the top of the stack. The third and fourth stack entries contain the other 32-bit number to be added.

* **At exit** - The top and second stack entries of the parameter stack contain a 32-bit signed double precision sum with the signed, most significant portion on the top of the stack.

D+ is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for D+ is D+ .



D+- (double \ single -- double)

D+- (pronounced "D-plus-minus") negates the sign of the double precision value in the second and third stack entries if the sign of the top single precision value is negative. The top value is then dropped.

The following truth table describes the outcome of all possible combinations:

SECOND ENTRY	TOP OF STACK	RESULT
+D2	+V1	+D2
+D2	-V1	-D2
-D2	+V1	-D2
-D2	-V1	+D2

DABS is an example of a word that uses D+- .

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value. The second stack entry contains the signed, high-order portion of a 32-bit double precision value. The third stack entry contains the low-order portion of this double precision value.

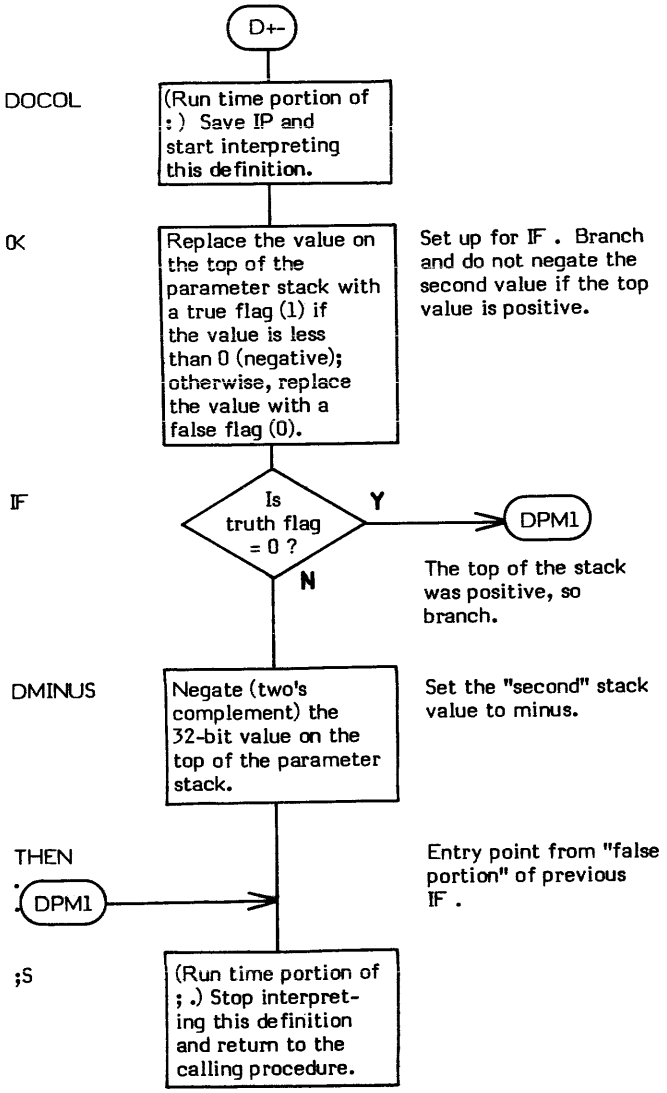
* **At exit** - The single precision value is dropped. The 32-bit double precision value is on the top of the parameter stack with the signed high order portion on the top of the stack. The low order portion is in the second stack entry.

D+- is a high level colon definition.

FORTH-79: There is no FORTH-79 equivalent for D+- .

Definition:

```
:  D+-  ( double \ single -- double )
     0<  IF  DMINUS  THEN  ;
```



D.

D. (double —)

D. (pronounced "D-dot") performs a binary-to-ascii conversion (pictured numeric output) on the 32-bit signed double precision value on the top of the stack and prints the result on the output device followed by one space.

The sign is only displayed if the value is negative.

D. does not pad with blanks. Use D.R if a specific minimum field length is needed.

The current value in BASE is used as the conversion radix.

The pictured numeric output words <# , #S , SIGN , and #> are used to actually perform the conversion. These are located in D.R which is used by D. .

* **At entry** - The top of the parameter stack contains a signed 32-bit double precision value to be converted and printed.

* **At exit** - No parameters.

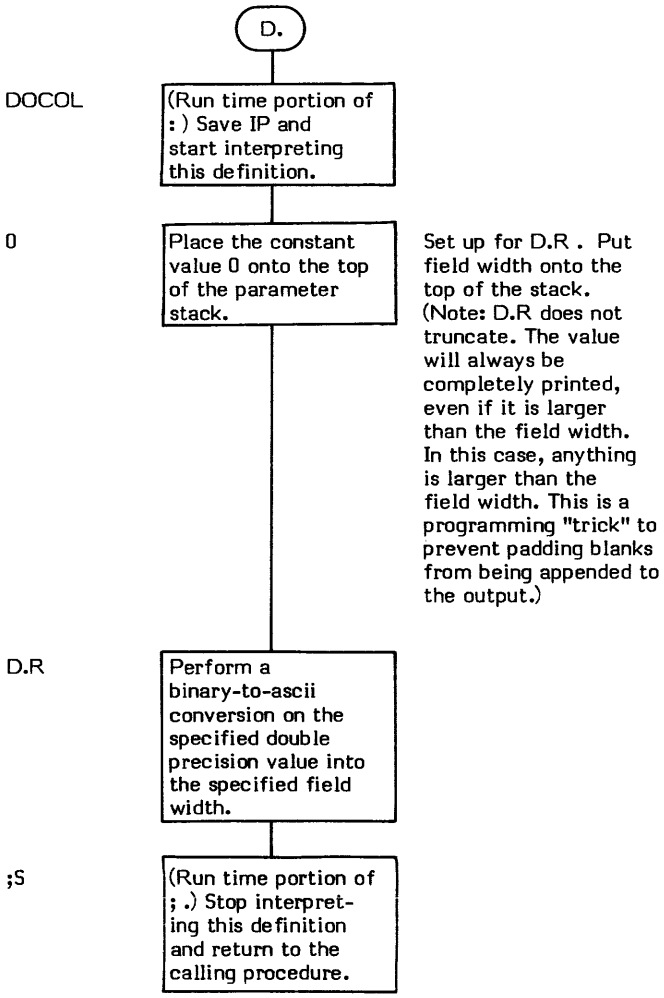
D. is a high level colon definition.

Refer to D.R .

FORTH-79: The FORTH-79 equivalent for D. is D. .

Definition:

```
:    D.    ( double -- )
        0 D.R    ;
```



D.R (double \ field width –)

D.R (pronounced "D-dot-R") performs a binary-to-ascii conversion (pictured numeric ouput) on a signed 32-bit double precision value and prints the result in a right justified field whose minimum width is specified by the value on the top of the stack. e.g., If a field length of 10 is specified and only 3 characters are printed the remainder of the field will be left padded with 7 blanks (␣) For example:

123 0 10 D.R results in ␣␣␣␣␣␣␣123

Note, however, that D.R is rather "stupid". If a field length of 3 characters is specified but the input value is large enough to print 10 characters all ten characters will be printed. i.e., D.R does not truncate values larger than the field width. For example:

123456 0 3 D.R results in 123456

No trailing blank is printed. Use D. if a minimum width field is not needed.

The current value in BASE is used as the conversion radix.

The sign is only displayed if the value is negative.

The basis of D.R is # . D.R is a good example of how to set up parameters for signed pictured numeric output.

D. is an example of a word which uses D.R .

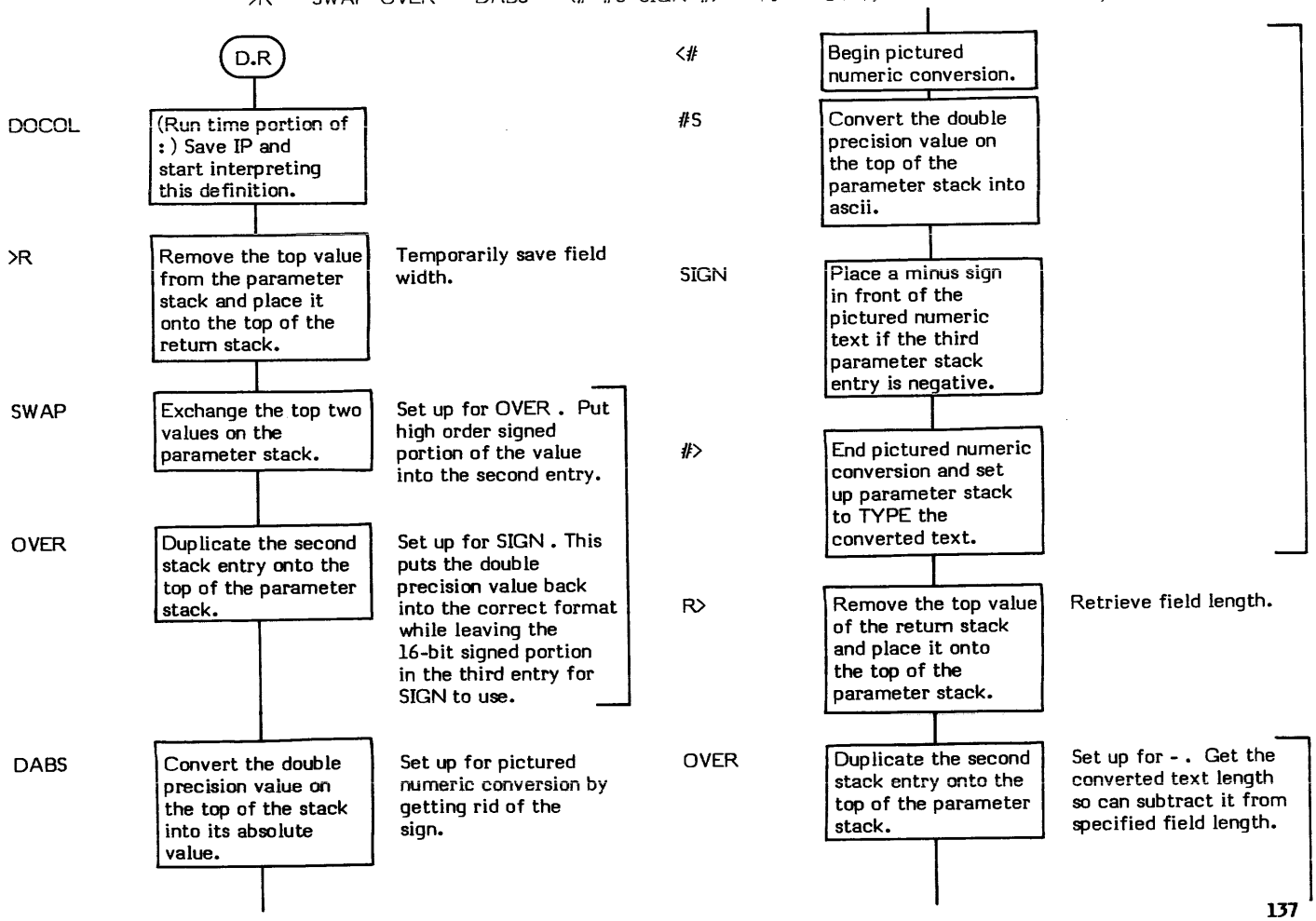
* **At entry** - The top of the parameter stack contains the signed 16-bit value which specifies the field width of the converted ascii string. The second and third stack entries contain a signed 32-bit double precision value to be converted and printed with the signed high order portion in the second stack entry and the low order portion in the third stack entry.
* **At exit** - No parameters.

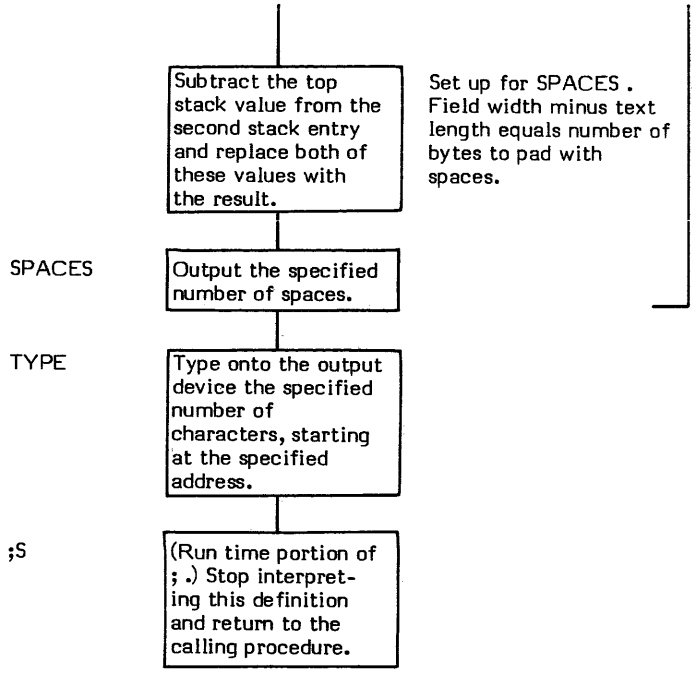
D.R is a high level colon definition.

Refer to <# , # , SIGN , #S , and #> .

FORTH-79: The FORTH-79 equivalent for D.R is D.R .

Definition: : D.R (double \ field width --)
>R SWAP OVER DABS <# #S SIGN #> R> OVER SPACES TYPE ;




Subtract the top stack value from the second stack entry and replace both of these values with the result.
Set up for SPACES . Field width minus text length equals number of bytes to pad with spaces.
SPACES
Output the specified number of spaces.
TYPE
Type onto the output device the specified number of characters, starting at the specified address.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

DABS (signed double value — absolute double value)

DABS (pronounced "D-ABS" for Double ABSolute) replaces a signed 32-bit double precision number with its absolute value. That is, negative values are made positive.

M/ is an example of a word which uses DABS .

* **At entry** - The top of the parameter stack contains the signed high order signed portion of a 32-bit double precision value. The second stack entry contains the low order portion of the value.

* **At exit** - The top of the parameter stack contains the high order portion of the absolute value of the original number. The second stack entry contains the low order portion of the value.

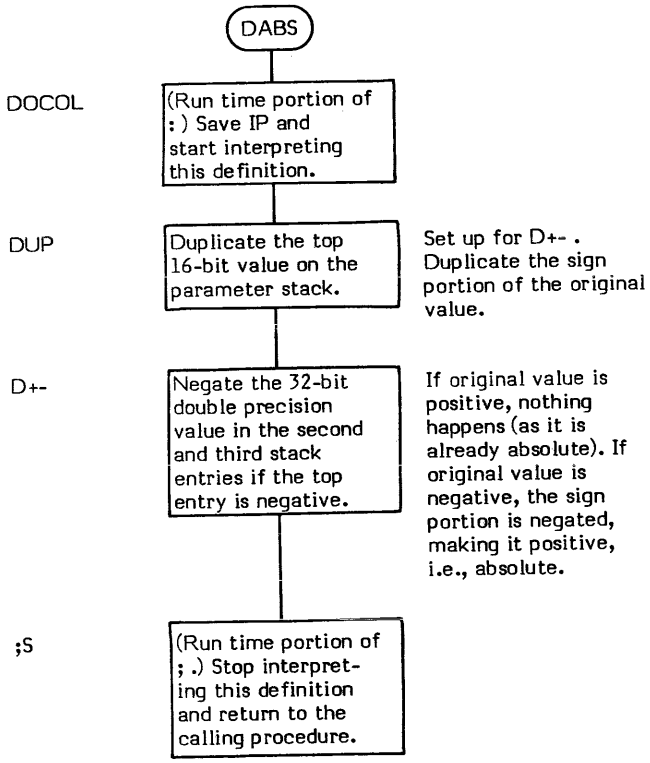
DABS is a high level colon definition.

Refer to D+- .

FORTH-79: The FORTH-79 equivalent for DABS is DABS .

Definition:

```
: DABS ( signed double -- absolute double )
    DUP D+- ;
```



DECIMAL

DECIMAL (—)

DECIMAL sets the user variable BASE to 10 (decimal). This causes all numeric input and output conversions to be performed in decimal (base 10).

Note that this is not an IMMEDIATE word.

* **At entry** - No parameters.
* **At exit** - No parameters.

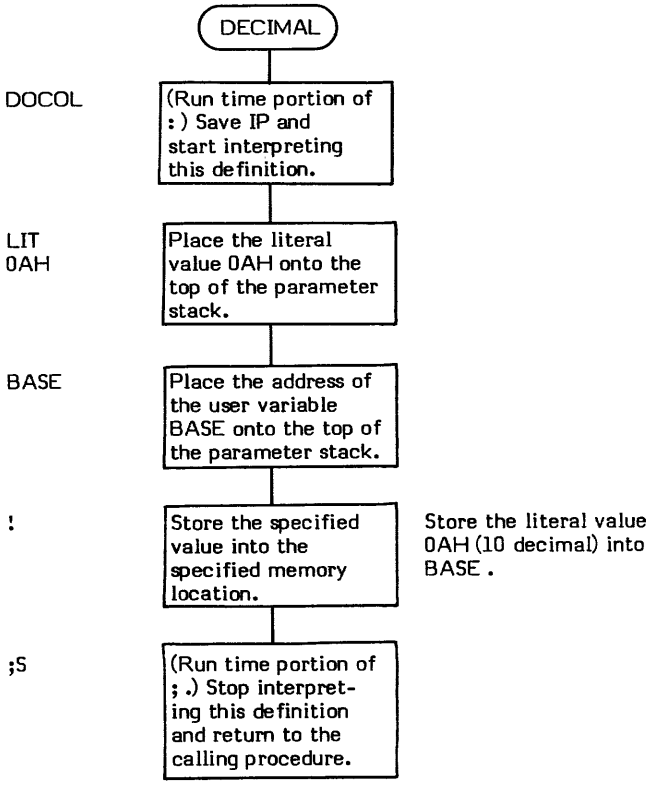
DECIMAL is a high level colon definition.

Refer to (NUMBER) , and BASE .

FORTH-79: The FORTH-79 equivalent for DECIMAL is DECIMAL .

Definition:

```
:  DECIMAL  ( -- )
     0A BASE !  ;
```



DEFINITIONS (--)

DEFINITIONS is used to specify the vocabulary into which new "definitions" are to be added (hence the name "DEFINITIONS").

Vocabularies in FORTH serve to limit the scope of a name; therefore, it is necessary to be able to specify which vocabulary a given name is to be appended. The user variable CURRENT points to the vocabulary which "currently" will have new definitions appended to it. DEFINITIONS copies the contents of the user variable CONTEXT into CURRENT. CONTEXT points to the vocabulary which is to be searched first when performing dictionary searches and is set to point to a specific vocabulary by stating (i.e., executing) the vocabulary name.

For example: Suppose a vocabulary named CAMERA has previously been created via the word VOCABULARY. Executing CAMERA sets CONTEXT to point to CAMERA (i.e, CAMERA will be searched first). Executing DEFINITIONS then copies CONTEXT into CURRENT (i.e, new definitions will now be added to CAMERA).

Normal usage would be in the form:

CAMERA DEFINITIONS

This not only causes definitions to be added to CAMERA but also helps produce highly readable, understandable FORTH listings describing exactly what is happening.

CREATE is the word which actually uses the data in CURRENT to append a new definition to a vocabulary. Refer to VOCABULARY for a more extensive explanation of vocabularies and the words which support them.

* **At entry** - No parameters but the user variable CONTEXT must contain a pointer to the vocabulary to which new definitions are to be added.
* **At exit** - No parameters.

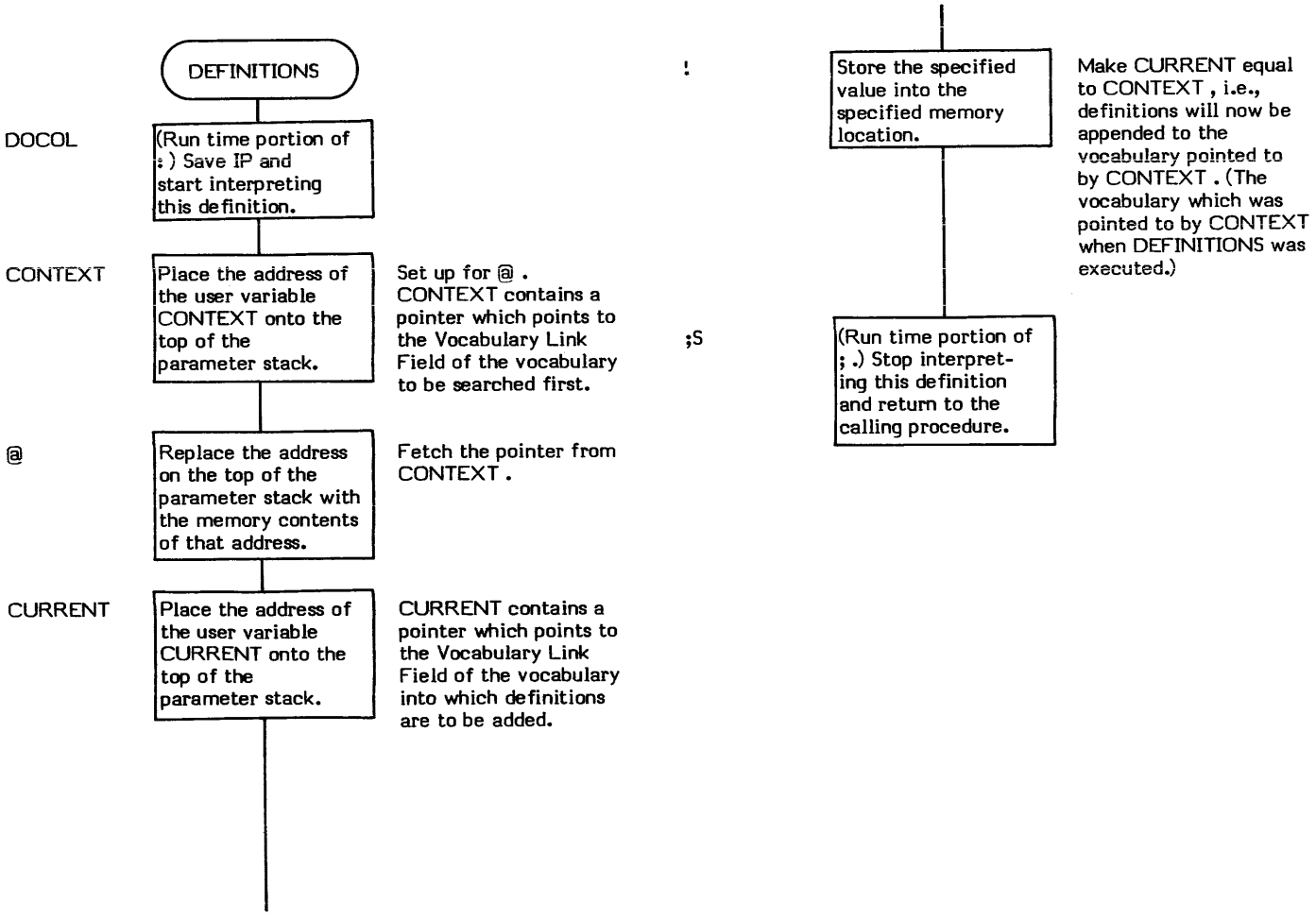
DEFINITIONS is a high level colon definition.

Refer to VOCABULARY , CURRENT , CONTEXT , and CREATE .

FORTH-79: The FORTH-79 equivalent for DEFINITIONS is DEFINITIONS .

Definition:

```
:  DEFINITIONS  ( -- )
     CONTEXT @ CURRENT !  ;
```



DIGIT

In Range - (char \ base — binary digit \ true flag)

Out of Range - (char \ base — false flag)

DIGIT converts an ascii character into its binary equivalent using the specified base value. Either the binary "digit" and a true flag are returned or just a false flag is returned depending upon DIGIT 's success.

Validity checking is performed such that a character whose original value is less than 30H (i.e., less than ascii "0") or between 3AH and 40H inclusive (greater than ascii "9" but less than ascii "A") is automatically invalid. After conversion from ascii to binary, any "digit" larger than the specified base is also invalid.

There is no ascii upper range validity test. The character may be whatever scale the specified base allows.

DIGIT works by first converting the ascii character to a binary value by stripping the ascii offset (30H) from the character. Then validity checking is performed and finally the value is tested to ensure it is within the maximum base range.

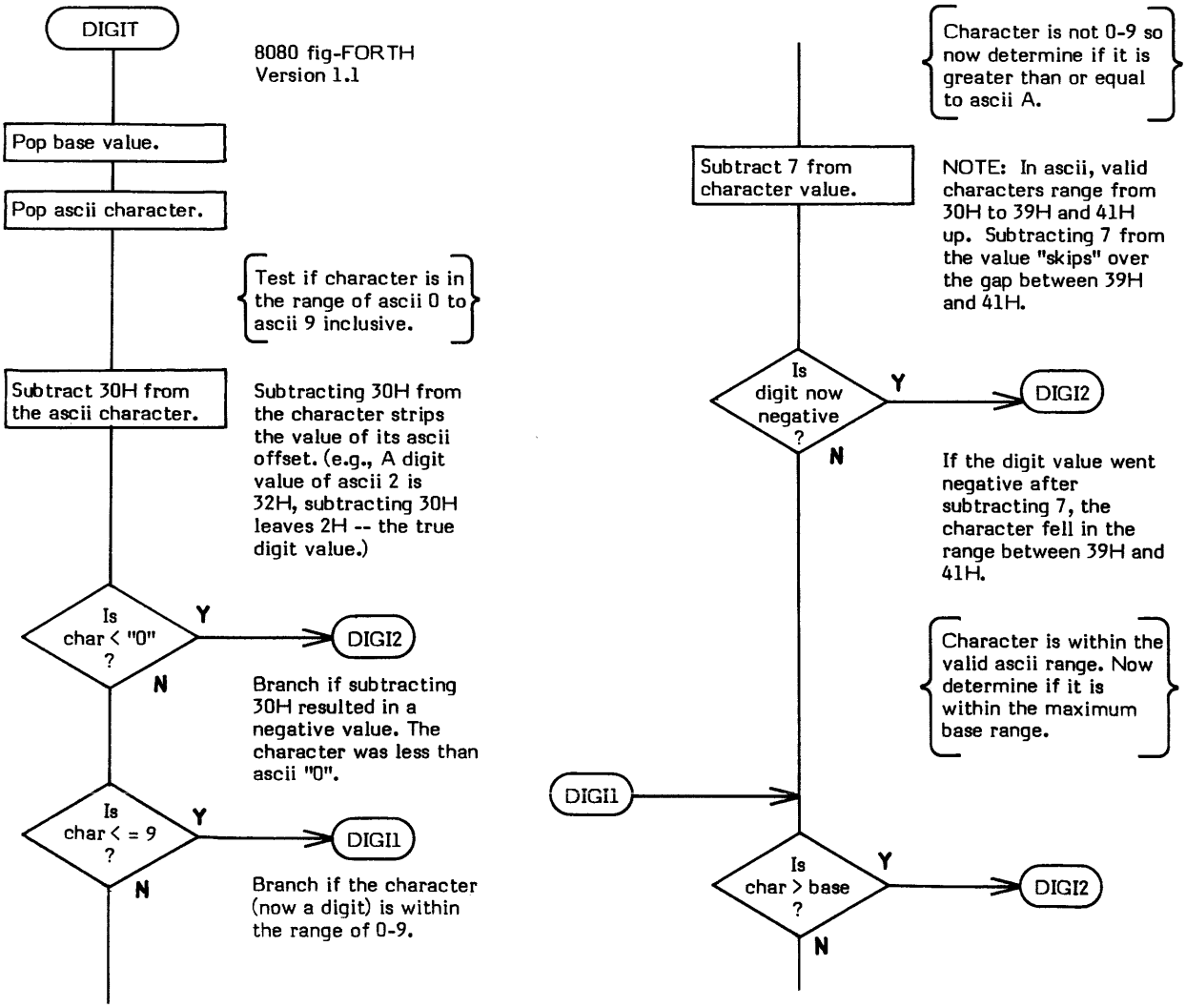
(NUMBER) is an example of a word which uses DIGIT .

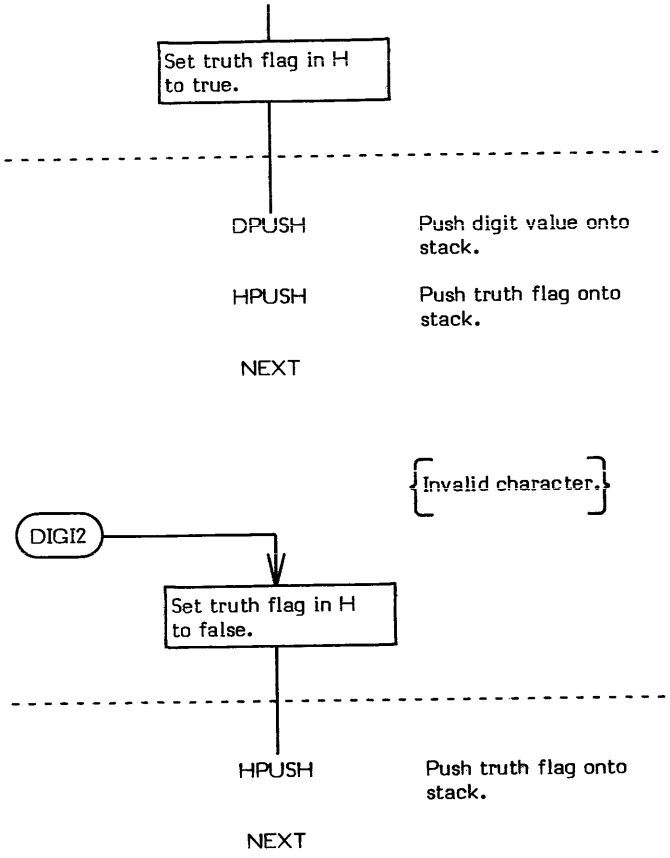
* **At entry** - The top of the parameter stack contains an unsigned 16-bit single precision value specifying the base value. The second stack entry contains an ascii character to be converted to a digit.
* **At exit (digit within range)** - The top of the parameter stack contains a "true" boolean truth flag. The second stack entry contains the binary "digit".
* **At exit (out of range)** - The top of the parameter stack contains a "false" boolean truth flag. No "digit" is returned.

DIGIT is a low level code primitive.

Refer to (NUMBER) .

FORTH-79: There is no FORTH-79 equivalent for DIGIT .




Set truth flag in H to true.
DPUSH
Push digit value onto stack.
HPUSH
Push truth flag onto stack.
NEXT
{Invalid character.}
DIGI2
Set truth flag in H to false.
HPUSH
Push truth flag onto stack.
NEXT

DLITERAL

DLITERAL

COMPILE TIME: (double —)
(Sequence 2)

EXECUTION TIME: (— double)
(Sequence 3)

DLITERAL is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

DLITERAL has no effect (and does not signal an error) if not executed within a colon definition.

The compile time action (Sequence 2) of DLITERAL is to compile a dynamically calculated 32-bit value into a definition. The end result of this compile action is the same as using LIT twice. The heart of DLITERAL is LITERAL, which in turn compiles LIT into the definition being compiled. The INTERPRETER compiles LIT into a definition "automatically" upon encountering a 16-bit numeric value in the input stream. However, if this numeric value is calculated dynamically during compilation, it will not exist in the input stream and therefore some other method must be used to compile LIT and the value into the dictionary. This is done by using the word LITERAL.

Since DLITERAL is concerned with a double precision value, LITERAL is used twice.

An example of the use of DLITERALwould be in dynamically calculating a maximum limit value to be used as a parameter. This calculation will be performed only once, at compile time, instead of each time the parameter is used. The definition would look like this:

```
: xxx [ #-OF-ENTRIES 2@  #-OF-PARMS 2@  D+ ]  DLITERAL  DOIT ;
```

[stops compilation, #-OF-ENTRIES and #-OF-PARMS contain double precision values which are fetched and then added together to form the double precision limit value which is left on the stack.] resumes compilation and DLITERAL compiles the 32-bit limit into the definition.

The execution time action of DLITERAL is that of LIT . That is, upon execution of each LIT , the compiled 16-bit value is placed onto the top of the parameter stack. NOTE that DLITERAL is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the signed high order word of a 32-bit double precision value. The second stack entry contains the low order portion of the value.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed high order word of a 32-bit double precision value previously compiled into the dictionary. The second stack entry contains the low order portion of the value previously compiled into the dictionary.

DLITERAL is a high level colon definition.

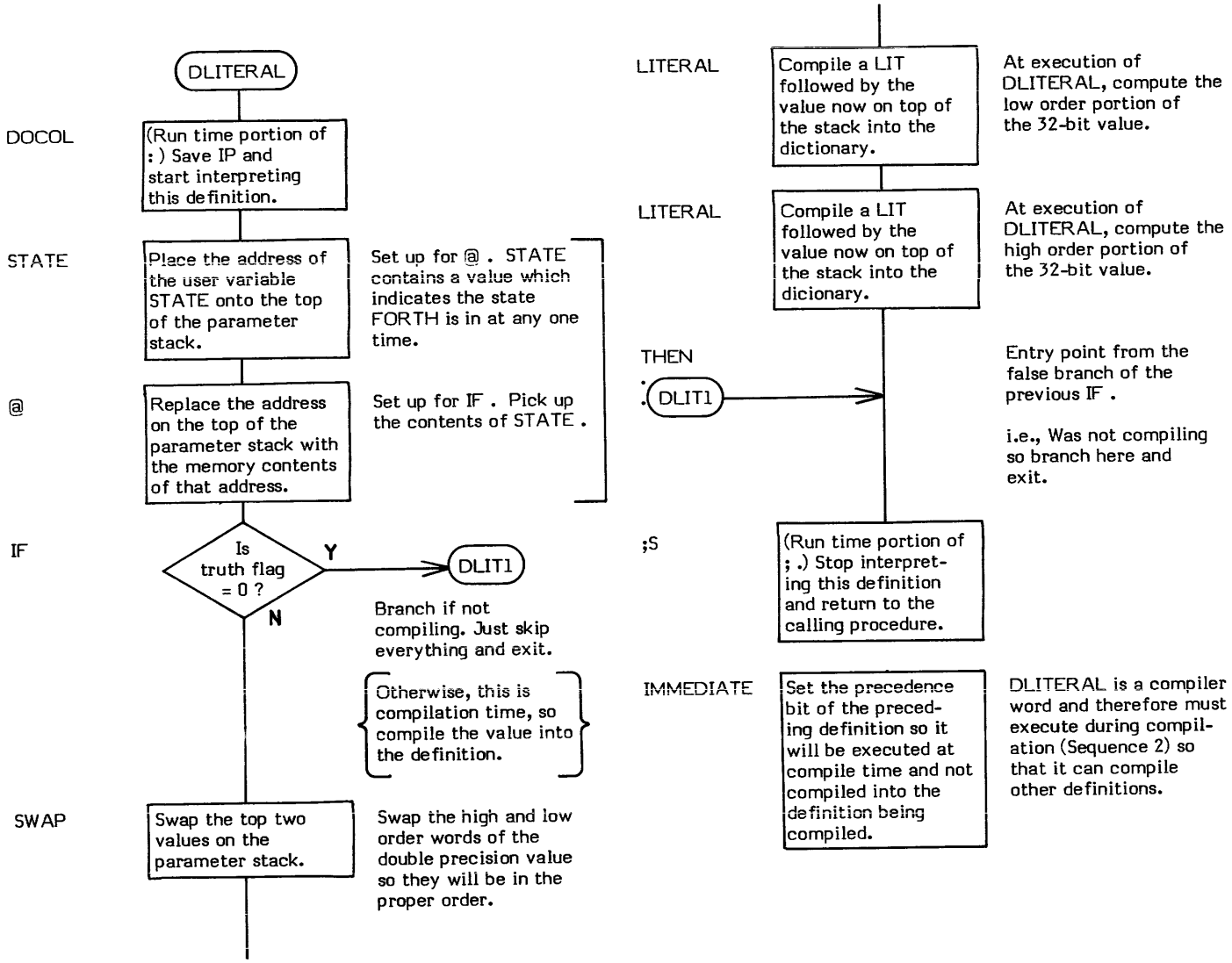
Refer to LITERAL , LIT , [,] , and INTERPRET .

FORTH-79: There is no FORTH-79 equivalent for DLITERAL .

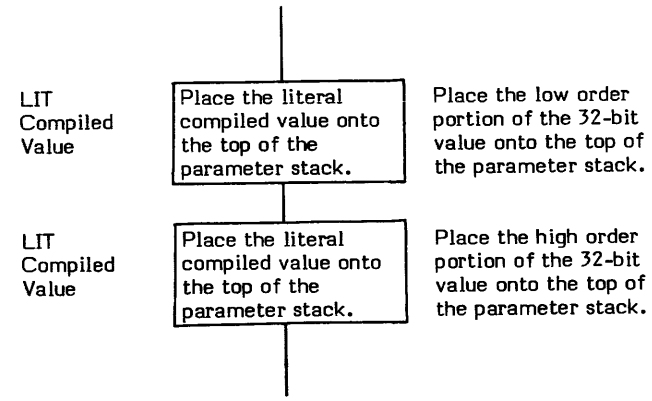
Definition:

```
:  DLITERAL  ( double -- )  ( compile time )
     STATE @  IF  SWAP  LITERAL LITERAL  THEN  ;  IMMEDIATE
```

COMPILE TIME action of DLITERAL (Sequence 2): (double —)



EXECUTION TIME action of DLITERAL (Sequence 3): (— double)



DMINUS

DMINUS (double -- negative double)

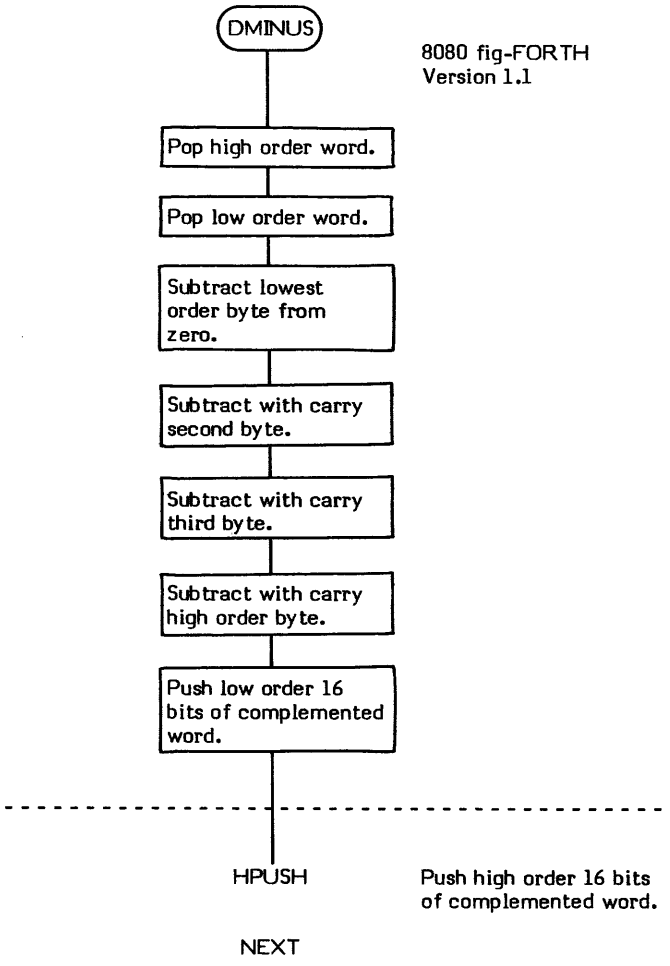
DMINUS (pronounced "D-minus" for Double MINUS) replaces the 32-bit signed value on the top of the parameter stack with its two's complement.

* **At entry** - The top and second words of the parameter stack contain a signed 32-bit double precision value with the signed, most significant portion on the top of the stack.

* **At exit** - The top and second entries of the parameter stack contain the two's complement of the given value with the signed, most significant portion on the top of the stack.

DMINUS is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for DMINUS is DNEGATE .



COMPILE TIME: (-- loop address\ 3)
(Sequence 2)

EXECUTION TIME: (Limit\ Index --)
(Sequence 3)

DO is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

DO is used to mark the beginning of a fixed repetition loop structure. This is identical in purpose to the FORTRAN DO and the BASIC FOR statements. DO is used to create a DO-LOOP in conjunction with LOOP or +LOOP in the form:

DO "loop body" LOOP

DO "loop body" (Increment Value on stack) +LOOP

The compile time action (Sequence 2) of DO is to compile (DO) into the definition and also to leave the entry point address of the next definition following (DO) on the parameter stack (i.e., the beginning address of the "loop body" portion of the loop.) This address is then used at compile time by LOOP or +LOOP to generate a branch offset. To provide compiler security, the value 3 is placed on the top of the parameter stack so that LOOP or +LOOP can check for it. This provides a somewhat secure (but not foolproof) method of checking for un-paired DO 's and LOOP 's.

The apparent execution time action (Sequence 3) of DO is actually performed by (DO) . This action is to remove the Index and Limit values from the parameter stack and place them onto the return stack to be used as inputs to LOOP or +LOOP . The Index (also referred to as Initial) value is incremented (or decremented) and compared with the Limit by LOOP or +LOOP each pass through the DO-LOOP .

The loop Index is accessible within the DO-LOOP structure by using the word I .

A " DO - LOOP " structure may be used only within a colon definition.

Refer to (DO) and I for a more detailed explanation of the execution time behavior of DO .

Note that DO is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

INDEX is an example of a word which uses DO .

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit single precision value 3. This value is checked by LOOP or +LOOP to provide compiler security. The second stack entry contains a 16-bit address pointing to the first definition to be executed as the "loop body" portion of the DO-LOOP structure.

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains the 16-bit signed Index value. The second stack entry contains the 16-bit signed Limit value.
* **At exit** - No parameters on the parameter stack; however, the return stack now has the Index value on the top of its stack, and the Limit value as the second entry on its stack.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

DO is a high level colon definition.

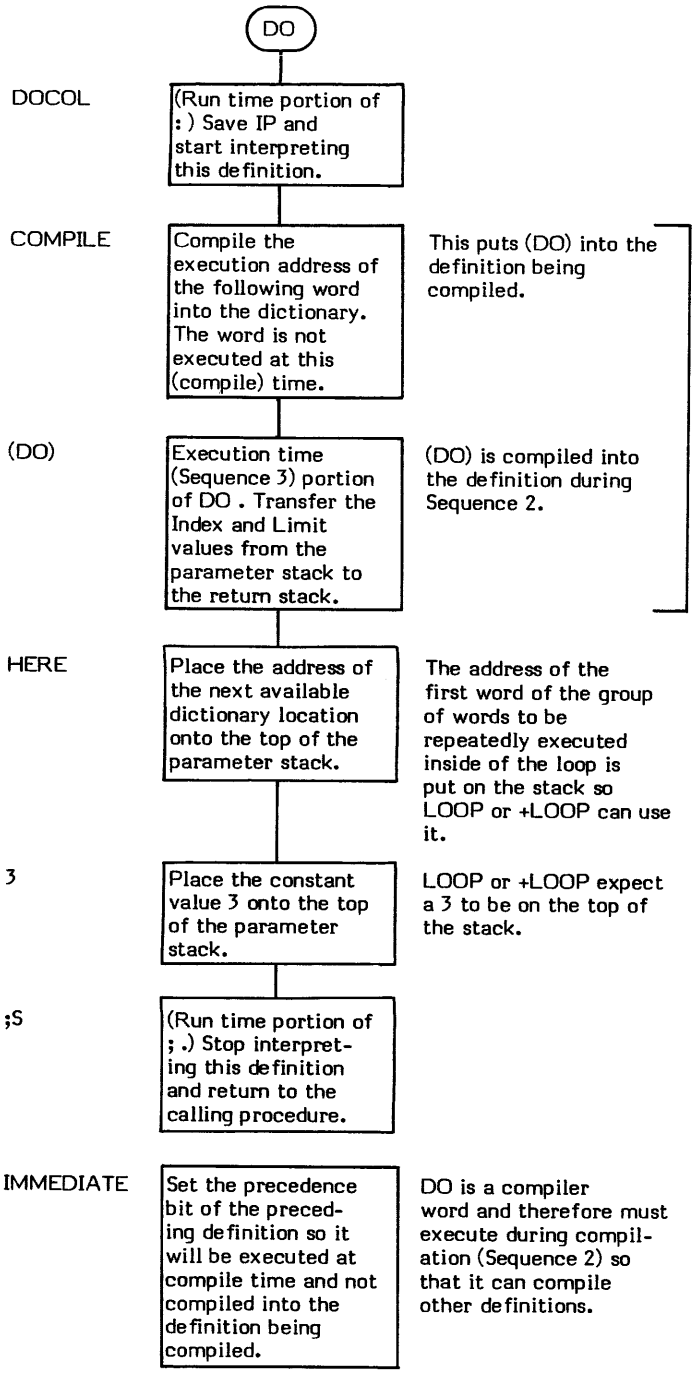
Refer to LOOP , (LOOP) , +LOOP , (+LOOP) , (DO) , and I .

FORTH-79: The FORTH-79 equivalent for DO is DO .

Definition:

```
:  DO   ( -- loop address\ 3 )   ( compile time )
     COMPILE (DO)   HERE   3   ;   IMMEDIATE
```

COMPILE TIME action of DO (Sequence 2): (— loop address \ 3)



EXECUTION TIME action of DO (Sequence 3): (Limit value \ Index value —)

Refer to (DO) for the EXECUTION TIME action of DO .

DOES>

DEFINITION TIME: (—)
(Sequence 1)

EXECUTION TIME: (second Parameter Field Address —)
(Sequence 2)

DOES> (pronounced "does") is a defining word and therefore exhibits two different sets of actions; those actions at definition time and those at execution time.

DOES> is a Sequence 1 defining word that is used to create Sequence 2 defining words. DOES> is normally used in conjunction with <BUILDS to create defining words ("parents") which in turn create other words ("children").

An overall explanation including examples of the use of <BUILDS and DOES> is included in the description of <BUILDS .

When creating a defining word (at Sequence 1), DOES> acts as a dividing line between the high level words that will create the "child" (at Sequence 2) and the high level words that are the "child's" run time procedure (at the "child's" Sequence 3).

;CODE is used to create code level run time procedures. See Figure DOES>-1.

The compile time action of DOES> is to overlay the "child" definition's Code Field with the address of the execution time code for DOES> . It also overlays the first Parameter Field location with the address of the definition following DOES> in the "parent" definition. This is the address of the "child's" first execution time procedure. When words that are then created with this defining word execute, the run time code of DOES> transfers control to the procedure following DOES> . (Refer to Figure DOES>-2.)

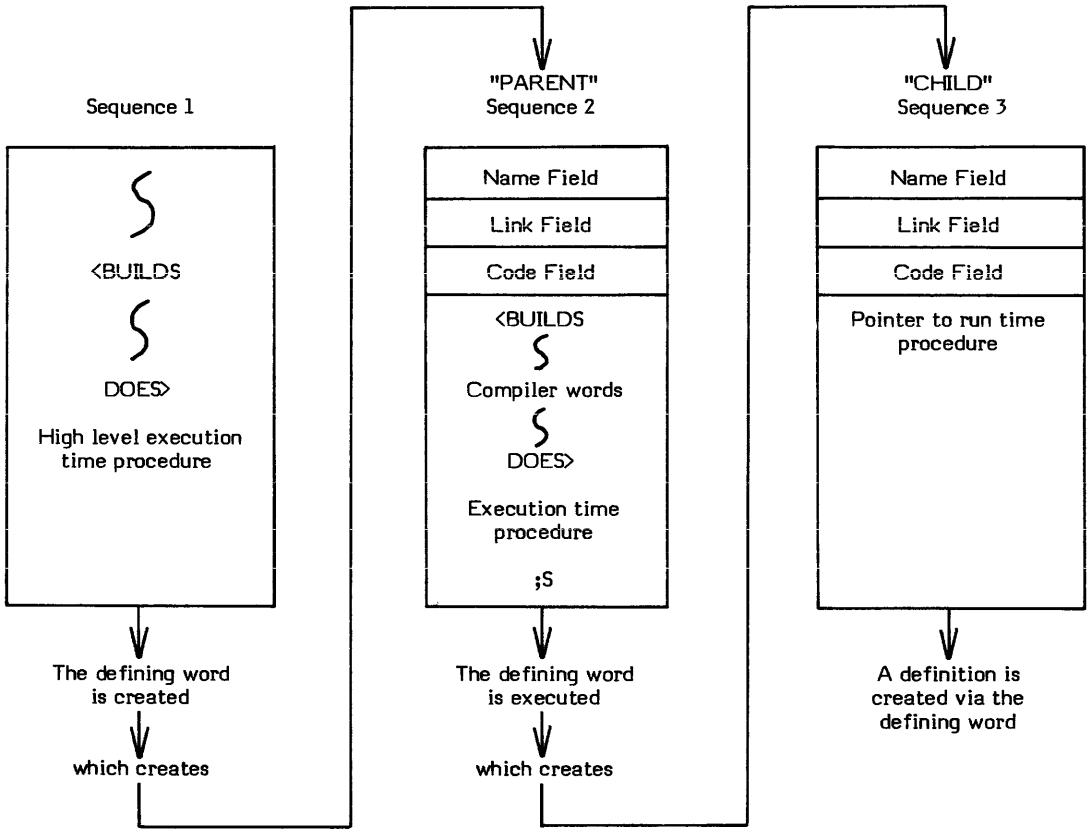
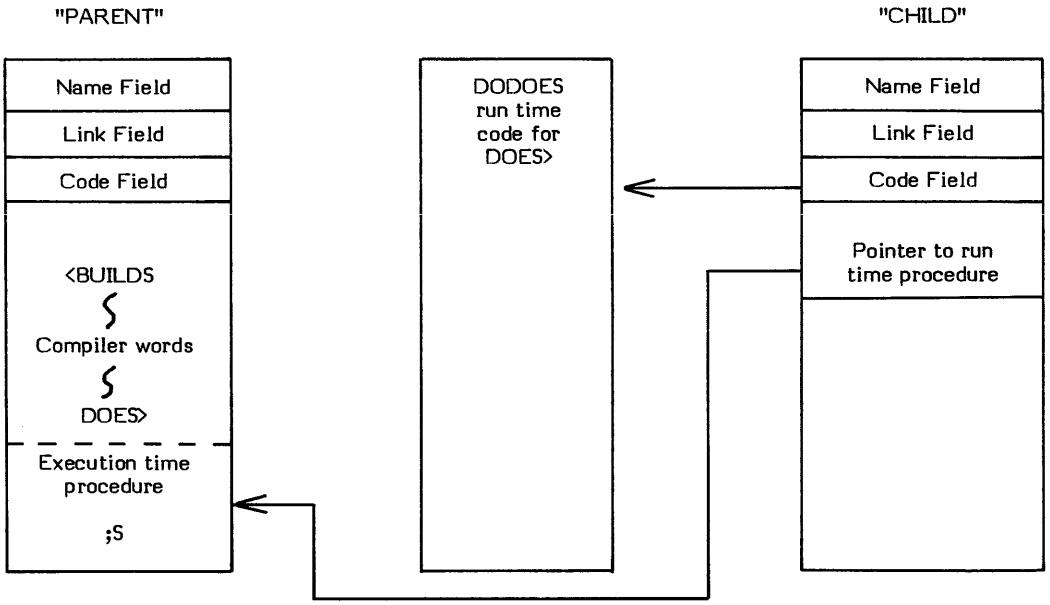


Figure DOES>-1

Compile Time Action of <BUILDS and DOES>



DODOES causes the run time procedure in the "parent" defining word to be executed when the "child" executes.

The "child's" execution procedure that resides in the "parent" is executed during Sequence 3.

Figure DOES>-2

Execution Time Action of DOES>

The execution time portion of DOES> is DODOES . This code performs two primary functions:

1. It places the address of the "child's" Parameter Field (actually the second entry in the Parameter Field) onto the top of the parameter stack so it will be available for the run time program.

2. It nests the return address of the "child" definition being executed and transfers control to the "child's" run time procedure (which is physically defined in the "parent" defining word definition).

Note that many "children" can be created via a single "parent" defining word and all of the "children" reference the same execution time (Sequence 3) procedure located within the "parent" definition.

In high level terms, the action of DOES> can be symbolically described as follows:

IP@ RP@ !	Push IP onto the return stack. i.e., Nest the address of the next word to be executed so the system can return after executing the run time procedure.
-2 RP +!	Increment the return stack pointer to point at the next available return stack location.
W@ 2+ IP!	W is now aiming at the Code Field of the "child" definition. Increment it by 2 so that W points to the pointer that aims at the run time procedure.
NEXT	Execute the procedure W is indirectly pointing to.

DOES> may only be used within a colon (:) definition.

VOCABULARY is an example of a word which uses DOES> .

DEFINITION TIME:

* **At entry** - No parameters.

* **At exit** -No parameters.

EXECUTION TIME (Sequence 3 execution time of the "child"):

* **At entry** - The top of the parameter stack contains the 16-bit address of the second Parameter Field entry in the "child's" definition. (Because the first Parameter Field entry was filled in by DOES> at compile time (Sequence 2), the second entry is the first location available to the definition.)

* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

DOES> is a high level colon definition.

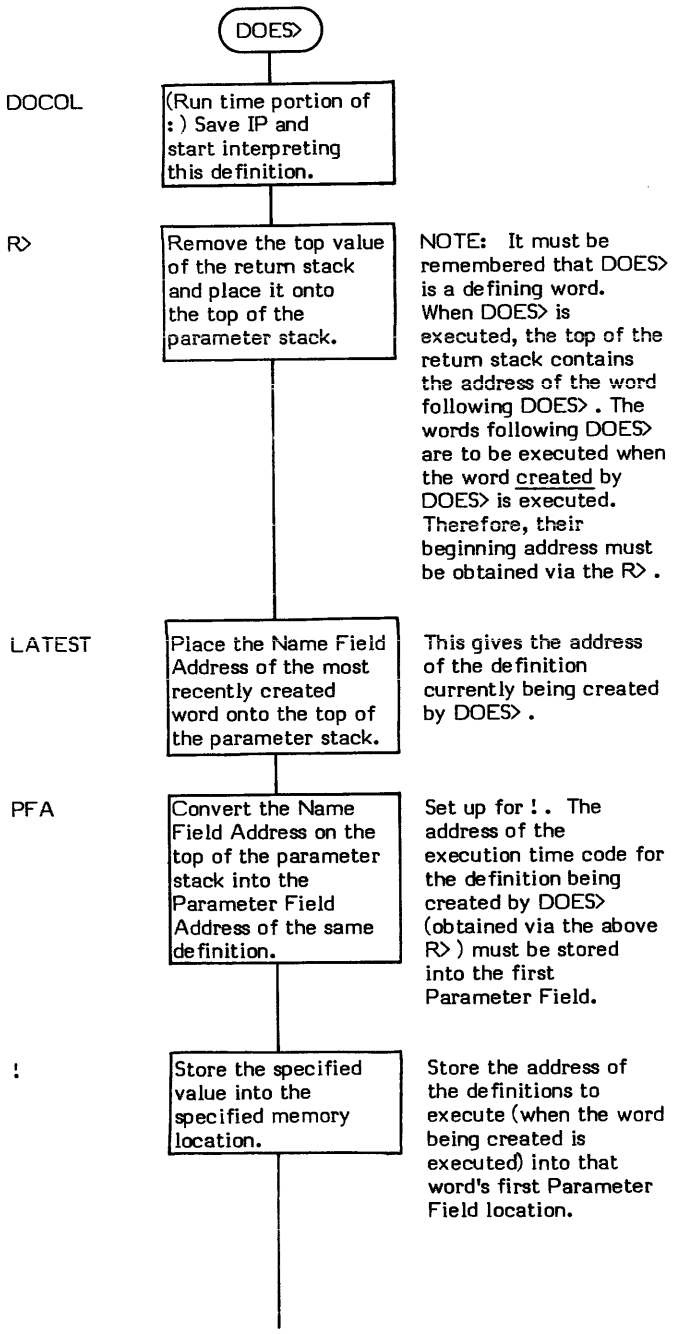
Refer to <BUILDS , ;CODE , and NEXT .

FORTH-79: The FORTH-79 equivalent for DOES> is DOES> .

Definition:
```
:  DOES>  ( -- )  ( compile time )
     R> LATEST PFA !  ;CODE
```
Note: the assembly language execution time code follows the ;CODE .

Note that the fig-FORTH Model version of DOES> uses the first word of the Parameter Field as a pointer to the execution time (Sequence 3) procedure. (This sometimes causes problems because PFA returns the true PFA, while the "definition's" Parameter Field actually begins one word later.) This problem is solved in FORTH-79. FORTH-79 DOES> does not utilize the first Parameter Field location. Also note that in FORTH-79, CREATE is paired with DOES> instead of <BUILDS .

DEFINITION TIME action of DOES> : (—)

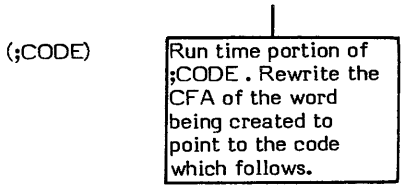


NOTE: It must be remembered that DOES> is a defining word. When DOES> is executed, the top of the return stack contains the address of the word following DOES> . The words following DOES> are to be executed when the word created by DOES> is executed. Therefore, their beginning address must be obtained via the R> .

This gives the address of the definition currently being created by DOES> .

Set up for ! . The address of the execution time code for the definition being created by DOES> (obtained via the above R>) must be stored into the first Parameter Field.

Store the address of the definitions to execute (when the word being created is executed) into that word's first Parameter Field location.



<BUILDS filled the Code Field of the word being created with the address of the run time code for CONSTANT . (;CODE) now overlays that address with the address of the run time code for DOES> . Note that at compile time (;CODE) eventually performs the same action as ;S .

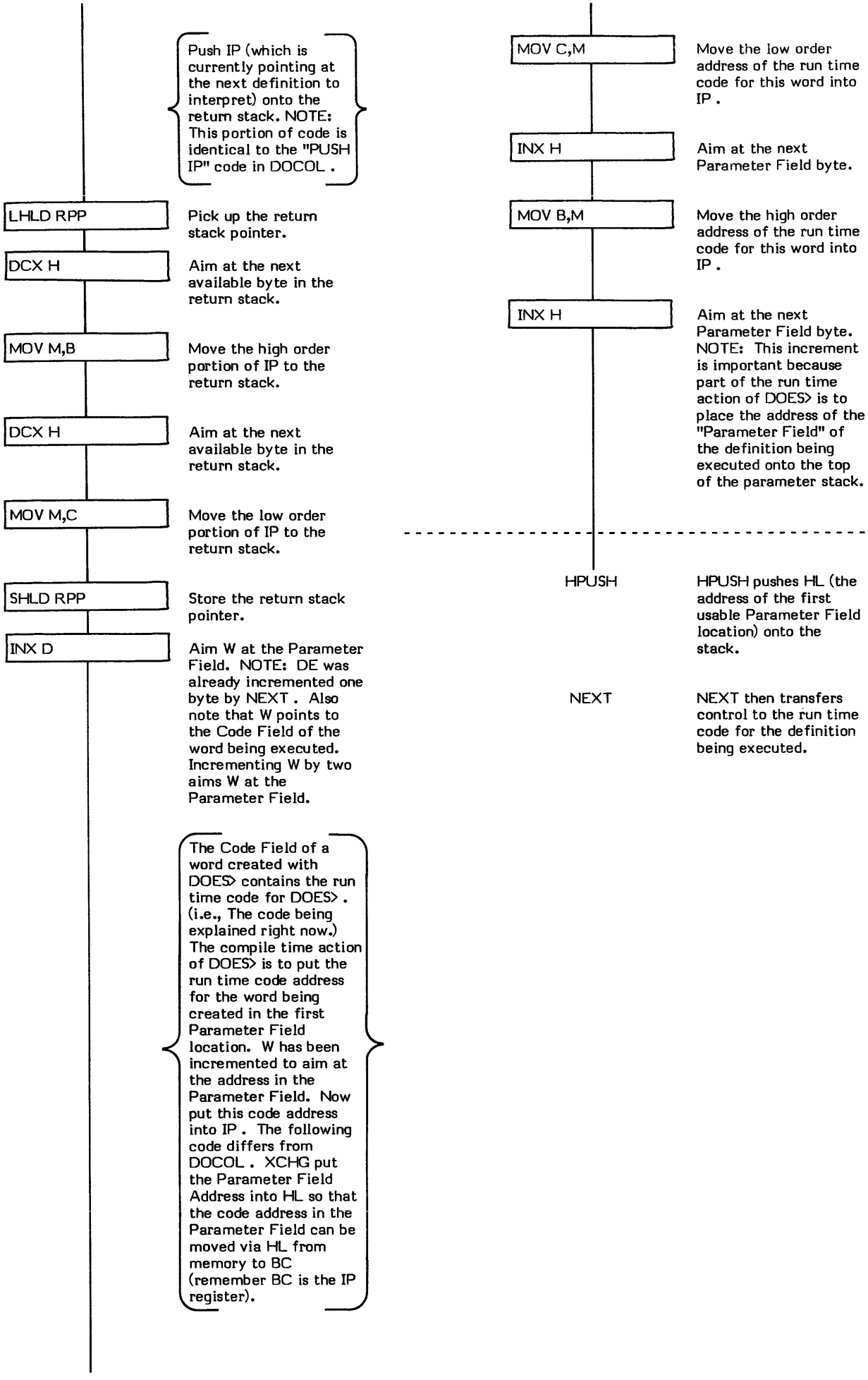
EXECUTION TIME action of DOES> : (addr —)

Note: This is a very machine dependent routine. The following is the 8080 fig-FORTH version 1.1 routine.

<BUILDS-DOES> is used in a compiling word. That compiling word is then used to create a new definition. (e.g., <BUILDS-DOES> creates VOCABULARY , VOCABULARY then creates FORTH .)

The following execution time code for DOES> is executed when the definition created by the compiling word (e.g., FORTH) executes.
NOTE: Register usage in the 8080 fig-FORTH Version 1.1 is:

BC = IP
DE = W
HL = Work Register


Push IP (which is currently pointing at the next definition to interpret) onto the return stack. NOTE: This portion of code is identical to the "PUSH IP" code in DOCOL .
LHLD RPP
Pick up the return stack pointer.
DCX H
Aim at the next available byte in the return stack.
MOV M,B
Move the high order portion of IP to the return stack.
DCX H
Aim at the next available byte in the return stack.
MOV M,C
Move the low order portion of IP to the return stack.
SHLD RPP
Store the return stack pointer.
INX D
Aim W at the Parameter Field. NOTE: DE was already incremented one byte by NEXT . Also note that W points to the Code Field of the word being executed. Incrementing W by two aims W at the Parameter Field.
The Code Field of a word created with DOES> contains the run time code for DOES> . (i.e., The code being explained right now.) The compile time action of DOES> is to put the run time code address for the word being created in the first Parameter Field location. W has been incremented to aim at the address in the Parameter Field. Now put this code address into IP . The following code differs from DOCOL . XCHG put the Parameter Field Address into HL so that the code address in the Parameter Field can be moved via HL from memory to BC (remember BC is the IP register).
MOV C,M
Move the low order address of the run time code for this word into IP .
INX H
Aim at the next Parameter Field byte.
MOV B,M
Move the high order address of the run time code for this word into IP .
INX H
Aim at the next Parameter Field byte. NOTE: This increment is important because part of the run time action of DOES> is to place the address of the "Parameter Field" of the definition being executed onto the top of the parameter stack.
HPUSH
HPUSH pushes HL (the address of the first usable Parameter Field location) onto the stack.
NEXT
NEXT then transfers control to the run time code for the definition being executed.

DP (— data address)

DP (pronounced "D-P" for Dictionary Pointer) is a user variable that contains an address pointer to the next available dictionary location. It is initialized during system startup by COLD with data from the origin parameter area.

The word HERE places the contents of DP onto the top of the parameter stack. The words ALLOT and FORGET alter the contents of DP .

The user variable DP is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used. Note that the word DP is not always a user variable. The physical location of DP is installation dependent (i.e., memory, a register, etc.).

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable DP .

Refer to COLD , HERE , ALLOT , FORGET , and USER .

FORTH-79: There is no FORTH-79 equivalent for DP .

DPL

DPL (— data address)

DPL (pronounced "D-P-L" for Decimal Point Location) is a user variable that reflects the number of digits found to the right of a decimal point when converting a numeric character string into a numeric value. (Actually, it is relative to the last decimal point encountered.) The words NUMBER and (NUMBER) perform this conversion. If a decimal point is not encountered, NUMBER leaves a -1 in DPL .

INTERPRET uses the contents of DPL to determine if a numeric value is to be treated as single precision (i.e., DPL contained -1 so no decimal point was encountered) or double precision (i.e., DPL did not contain -1 so a decimal point was encountered). Although INTERPRET uses a decimal point only for purposes of determining precision, this does not mean that an application cannot use the value stored in DPL to process the decimal point location.

The user variable DPL is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.

* **At exit** - The top of the parameter stack contains the address of the user variable DPL .

Refer to NUMBER , (NUMBER) , INTERPRET , and USER .

FORTH-79: DPL is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set". Note that DPL as described in the Reference Word Set is used for numeric output, not input.

DPUSH (— value)

DPUSH is strictly an 8080 fig-FORTH Version 1.1 inner interpreter routine entry point.

Entry at this entry point causes the 16-bit contents of the DE register pair to be pushed onto the top of the parameter stack. Then the contents of the HL register pair is also pushed onto the top of the parameter stack then NEXT is executed.

Refer to NEXT .

DR0 DR1

DR0 (–)

DR1 (–)

DR0 and DR1 are standard FORTH mass storage device selection words. These words store a block number offset into the user variable OFFSET , so that the programmer can reference a selected device as if the block numbers for that device begin with block zero.

All FORTH mass storage I/O is virtual I/O. That is, desired information is referenced via block numbers, which to the programmer, are synonymous to the memory address of the desired data. The type of mass storage, the hardware addresses, and the media are normally transparent to the programmer.

Each storage device on the system has a unique range of block numbers assigned to it. This allows device selection to be based solely upon block number range. For example, Disk Drive 1 may contain blocks 0 through 799; while Disk Drive 2 may contain blocks 800 through 1199.

This explicit addressing does pose a slight problem, however. For instance, it is difficult to remember that certain data on a diskette resides at block 160 when the diskette in Drive 1; but the same data resides at block 960 when the diskette is in Drive 2.

BLOCK solves this problem through the use of OFFSET . Immediately upon entry, the contents of OFFSET is added to the desired block number. This allows implicit drive selection by previously setting (i.e., biasing) OFFSET to the starting block number of the selected drive. Explicit drive selection may be performed by ensuring that OFFSET contains the value 0 before calling BLOCK .

The purpose of drive selection words are to fill the user variable OFFSET with a bias value equal to the starting block number of the desired device. That way (for identical devices) a particular block of data can always be referenced via the same block number.

Note that device selection words are not limited to DR0 and DR1 . Another common, but not standard, selection word is MT0 which is used to select a tape drive. Device selection words are obviously installation dependent.

* **At entry** - No parameters.
* **At exit** - No parameters.

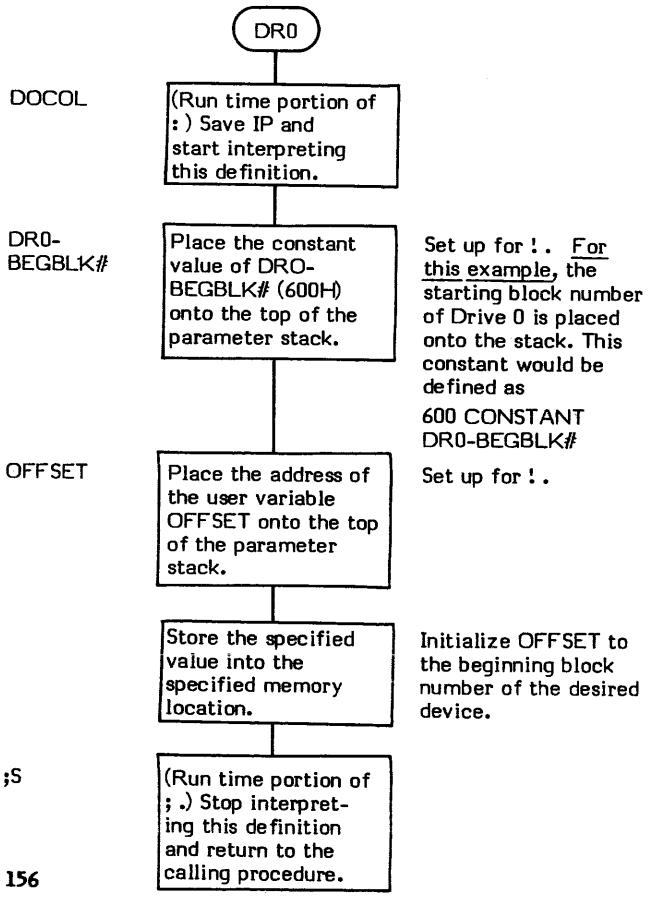
DR0 and DR1 are high level colon definitions.

Refer to BLOCK , R/W , and OFFSET .

FORTH-79: There is no FORTH-79 equivalent for DR0 or DR1 .

Definition:

```
:  DR0  ( – )
       DR0-BEGBLK#  OFFSET  !  ;
```



DROP **(value to be dropped —)**

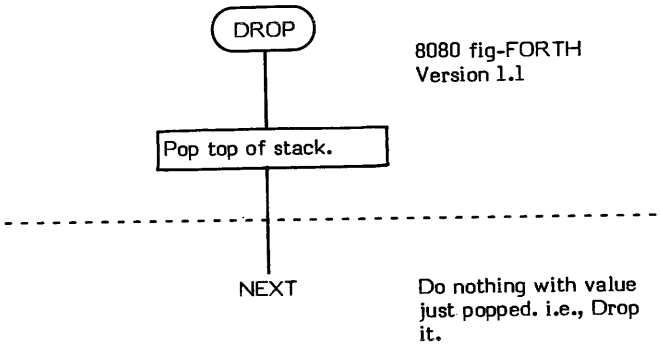
DROP discards the 16-bit value presently on the top of the parameter stack.

DROP is commonly used to discard parameters from the stack before exiting a word or procedure. #> is an example of a word which uses DROP .

* **At entry** - The top of the parameter stack contains the 16 bit value to be discarded.
* **At exit** - This value has been discarded (dropped).

DROP is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for DROP is DROP .



DUP

DUP (value 1 -- value 1 \ value 1)

DUP (pronounced "dupe") duplicates the 16-bit value presently on the top of the parameter stack.

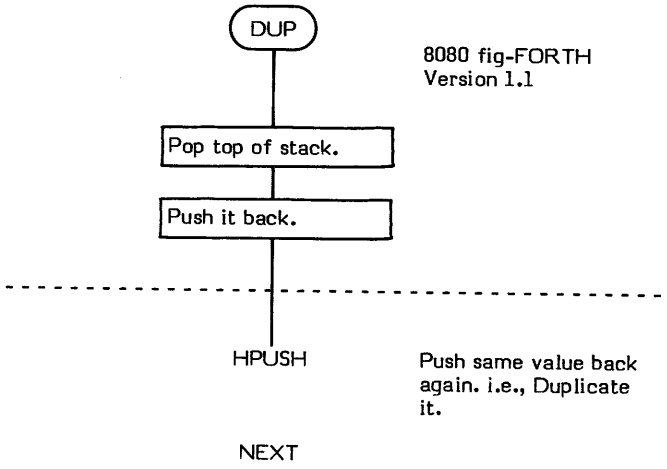
DUP is a very commonly used word. It is usually used to make a copy of an input parameter so the original value will still be available after an intermediate word "uses up" the input parameter.

COUNT is an example of a word which uses DUP .

* **At entry** - The top of the parameter stack contains the 16-bit value to be duplicated.
* **At exit** - The first and second 16-bit stack values are equal.

DUP is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for DUP is DUP .



ELSE

COMPILE TIME (Sequence 2): (offset address \2 -- offset address \2)

EXECUTION TIME (Sequence 3): (--)

ELSE is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

ELSE is used as the beginning of the "false portion" of an IF structure in the form:

IF "True Portion" ELSE "False Portion" THEN

(Beware -- this differs in form from the PASCAL IF-THEN-ELSE structure.) The use of ELSE in this type of structure is optional. If ELSE is omitted, a false boolean input causes control to simply branch to the word immediately following the THEN . When used, ELSE must be located within an IF-ELSE-THEN structure. (ENDIF may be used in place of THEN .)

An IF-ELSE-THEN structure must be used within a colon definition.

The compile time action (Sequence 2) of ELSE is to compile a BRANCH into the definition being compiled. (This BRANCH is what divides the structure into two parts. After the "true portion" executes, the BRANCH is encountered and control is passed around the "false portion" to the word immediately following the THEN .) The location following BRANCH is reserved for the branch address. This location address is then placed on the parameter stack so that it may be later resolved by THEN . Finally, the branch offset for the 0BRANCH in the preceding IF statement is resolved by executing an ENDIF or THEN . If the ELSE is left out of the IF structure, the terminator THEN (or ENDIF) resolves the offset.

Some compiler security is provided by checking for, and also leaving, a 2 on the top of the stack. An ELSE without a preceding IF will probably not encounter a 2 on the top of the stack. (NOTE: During compilation, IF leaves a 2 on the top of the stack.) This is not foolproof however. An ELSE followed by another ELSE will not be detected by checking for a 2 since ELSE also leaves a 2.

The run time action (Sequence 3) of ELSE is to prevent execution of the "true portion" of the structure from proceeding sequentially on into the "false portion" of the structure. At compile time, the branch offset of the 0BRANCH in the IF was resolved by ELSE to point to the "false portion". A false boolean flag (0) input to the IF will cause this branch to be taken and the "false portion" will be executed.

Note that ELSE is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 2 used for compiler security. The second stack entry contains a 16-bit address specifying the memory location where the offset is to be stored. (This address is also used to compute the offset itself.)
* **At exit** - The top of the parameter stack contains the 16-bit signed single precision value 2 used for compiler security. The second stack entry contains a 16-bit address specifying the memory location where the offset is to be stored by the terminating THEN or ENDIF . (This address is also used to compute the offset itself.)

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

ELSE is a high level colon definition.

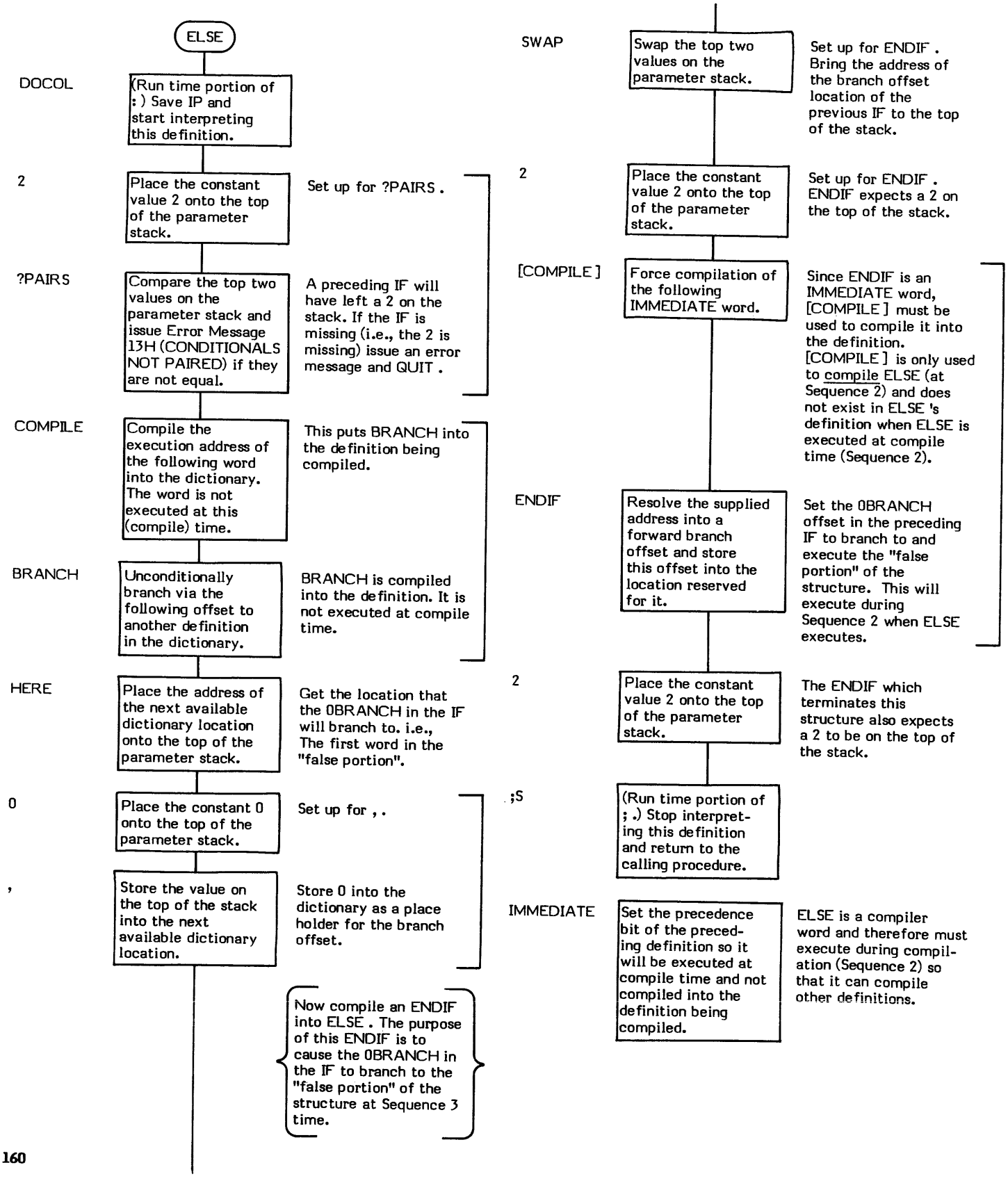
Refer to IF , THEN , ENDIF , and BRANCH .

FORTH-79: The FORTH-79 equivalent for ELSE is ELSE .

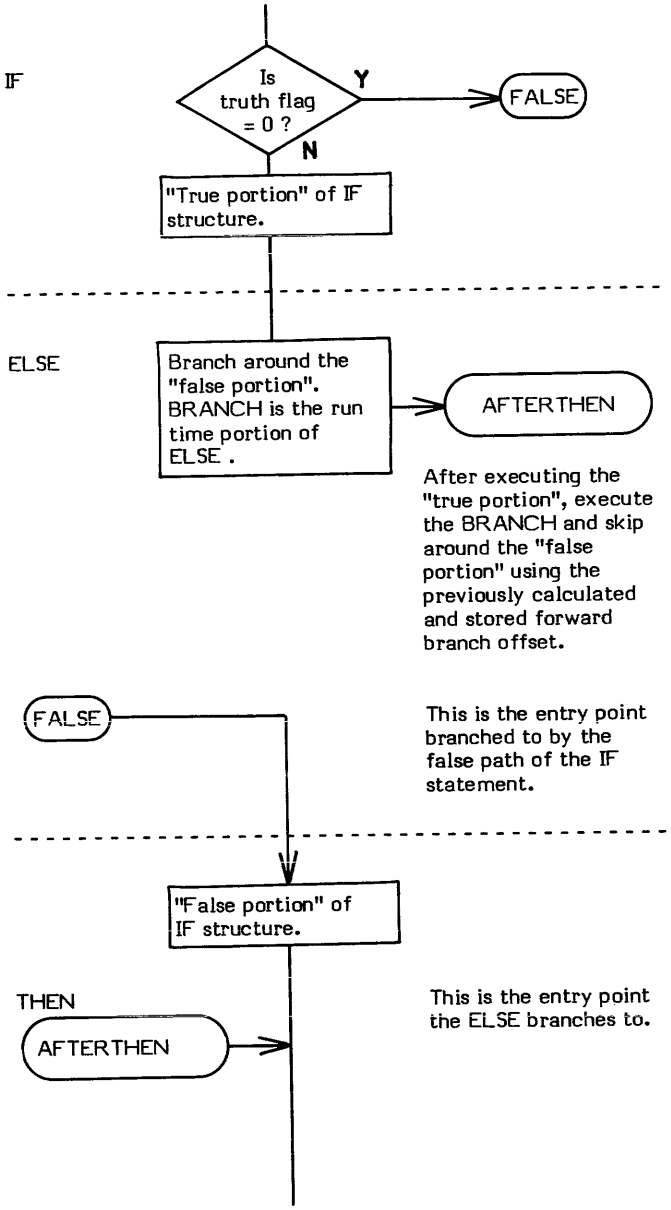
Definition:

```
:  ELSE  ( offset address\ 2 -- offset address\ 2 )  ( compile time )
       2 ?PAIRS  COMPILE BRANCH  HERE 0 ,  SWAP 2
       COMPILE ENDIF  2  ;  IMMEDIATE
```

COMPILE TIME action of ELSE (Sequence 2): (offset address \ 2 — offset address \ 2)



EXECUTION TIME action of ELSE (Sequence 3): (—)



EMIT

EMIT (character to EMIT --)

EMIT outputs the character on the top of the stack to the output device.

TYPE is an example of a word which uses EMIT .

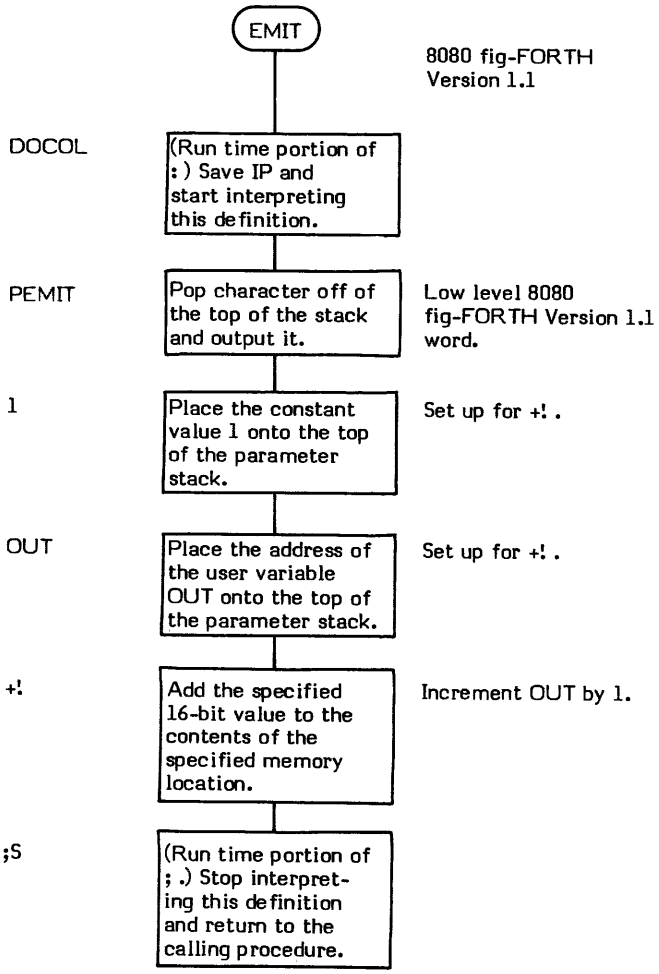
* **At entry** - The top of the parameter stack contains a 16-bit value of which the low order 8-bits are to be output.
* **At exit** - No parameters.

EMIT is a high level colon definition.

Refer to OUT .

FORTH-79: The FORTH-79 equivalent for EMIT is EMIT .

Definition: : EMIT (character --)
PEMIT 1 OUT +! ;



EMPTY-BUFFERS (–)

EMPTY-BUFFERS erases the contents of all of the buffers to zero. All update bits are erased to zero. Note that the null terminating word at the end of each buffer is also reset to zero. (Refer to +BUF for a detailed description of the buffers.)

EMPTY-BUFFERS can be used to prevent modified (and subsequently "updated" data) from being written and therefore changing data already on disk.

The word is also used just after system start-up, prior to any disk accesses. This is done because at start-up time, the buffers are un-initialized and contain random data. It is possible that the update bit of a buffer may be randomly set on and the block number may be a valid block number. If this happened, random data would be written over a valid block on disk whenever BUFFER first allocated that buffer.

The name EMPTY-BUFFERS is purposely made rather cumbersome to type to prevent accidental execution because of its potentially destructive power.

* **At entry** - No parameters.
* **At exit** - No parameters.

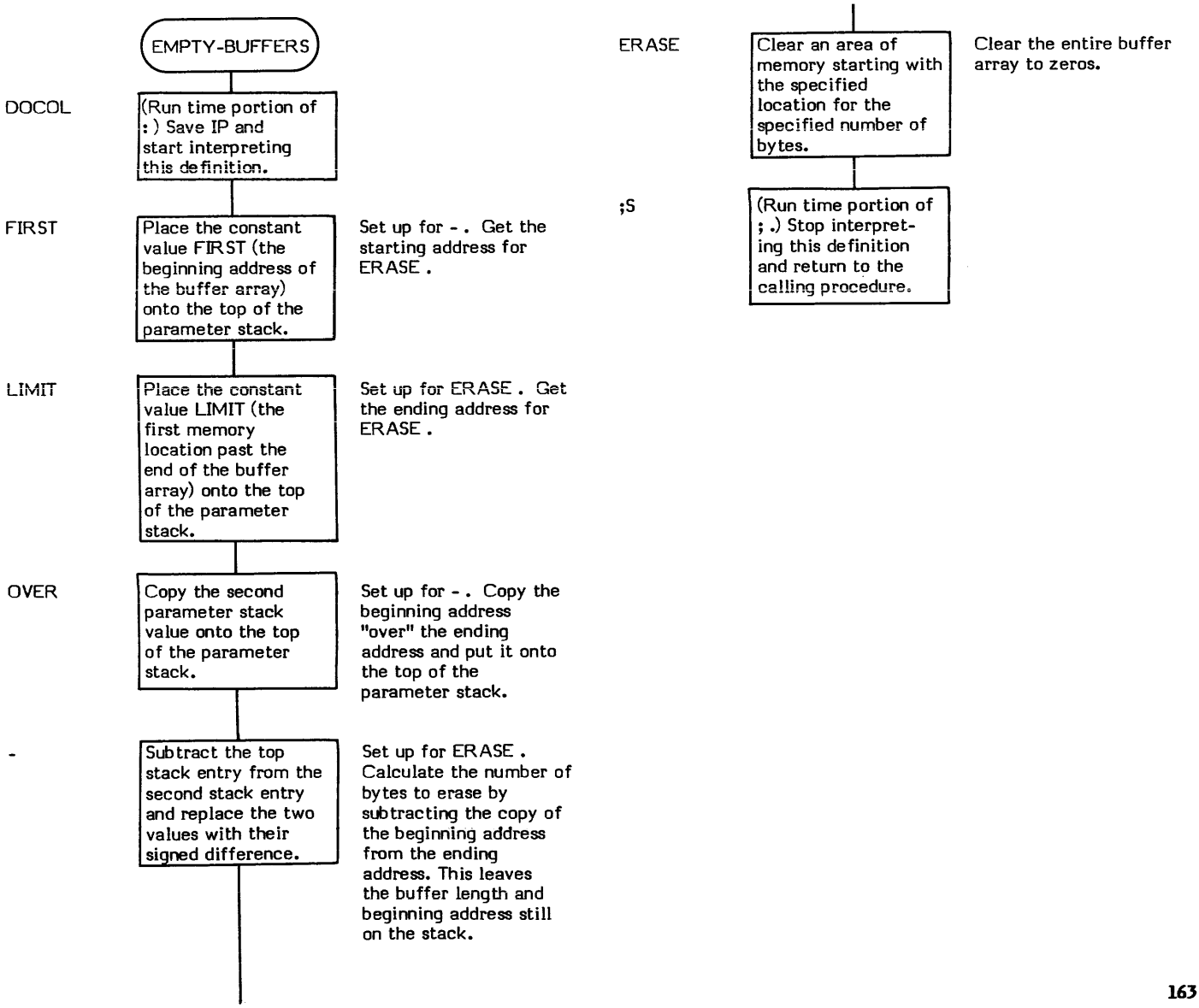
EMPTY-BUFFERS is a high level colon definition.

Refer to +BUF , UPDATE , and FLUSH .

FORTH-79: The FORTH-79 equivalent for EMPTY-BUFFERS is EMPTY-BUFFERS .

Definition:

```
:  EMPTY-BUFFERS  ( -- )
       FIRST LIMIT OVER - ERASE  ;
```



ENCLOSE

ENCLOSE (beg string addr \ offset to 1st non-delim \ offset to 1st delim \ offset to 1st unexamined char)

ENCLOSE performs a parsing function upon a given text string. Given a beginning text address and a delimiter character, ENCLOSE scans the text and leaves its results on the stack. These results are in the form of byte offsets from the beginning of the text address. The parameters returned by ENCLOSE are:

1. An offset to the beginning of a delimited string (i.e., the first non-blank character).
2. The offset to the end of that same string.
3. The offset to the next byte to examine.
4. The original beginning address.

This primitive is normally used by WORD to parse the input data stream into individual words delimited by blanks. Note: To avoid any confusion, the "text string" parsed by ENCLOSE is a pure character string, not characters preceded by a length byte. (Actually, WORD creates the length byte using the offsets returned by ENCLOSE .)

* **At entry** - The top of the parameter stack contains a 16-bit value of which the low order 8-bits are the delimiter character. The second stack entry contains the 16-bit beginning address (inclusive) of the text string to be parsed.
* **At exit - The general form of the parameter stack upon exit is:**

Top of stack -	Offset (relative to specified beginning address) to first unexamined character (i.e., character at which to start next scan).
Second entry -	Offset to the 1st delimiter after text.
Third entry -	Offset to the first non-delimiter character.
Fourth entry -	Beginning address of character string to parse as specified at entry.

There are three specific conditions that may arise when parsing any given character string. The following is a description of each condition and a description of the stack upon exit from that condition:

CONDITION 1 - Scan encountered a null character before encountering any non-delimiter characters.

Condition of stack at exit:

Top of stack - Offset to null.

Second entry - Offset to byte following null.

Third entry - Offset to null.

Fourth entry - Beginning address of character string to examine.

CONDITION 2 - Scan enountered non-delimiter character(s) terminated by a null.

Condition of stack at exit:

Top of stack - Offset to null.

Second entry - Offset to null.

Third entry - Offset to first non-delimiter character.

Fourth entry - Beginning address of character string to examine.

CONDITION 3 - Scan encountered non-delimiter characters terminated by a delimiter character.

Condition of stack at exit:

Top of stack - Offset of first unexamined character.

Second entry - Offset to first delimiter after non-delimiter character(s).

Third entry - Offset to first non-delimiter character.

Fourth entry - Beginning address of character string to examine.

NOTE: The fig-FORTH 8080 Version 1.1 of ENCLOSE cannot correctly parse undelimited text strings with a length of more than 255 bytes.

ENCLOSE is a low level code primitive.

Refer to WORD .

FORTH-79: There is no FORTH-79 equivalent for ENCLOSE .

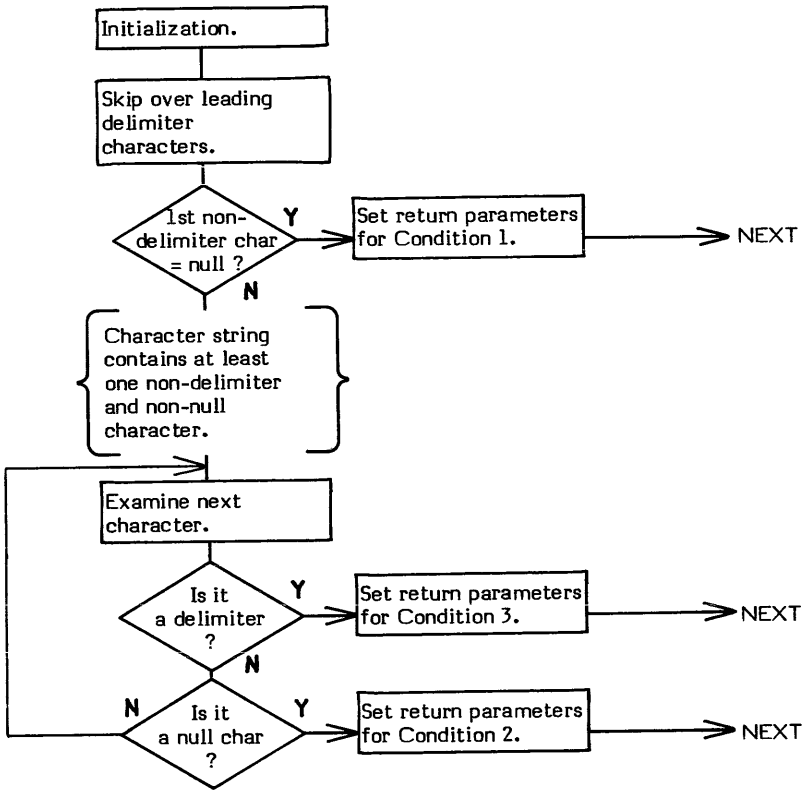
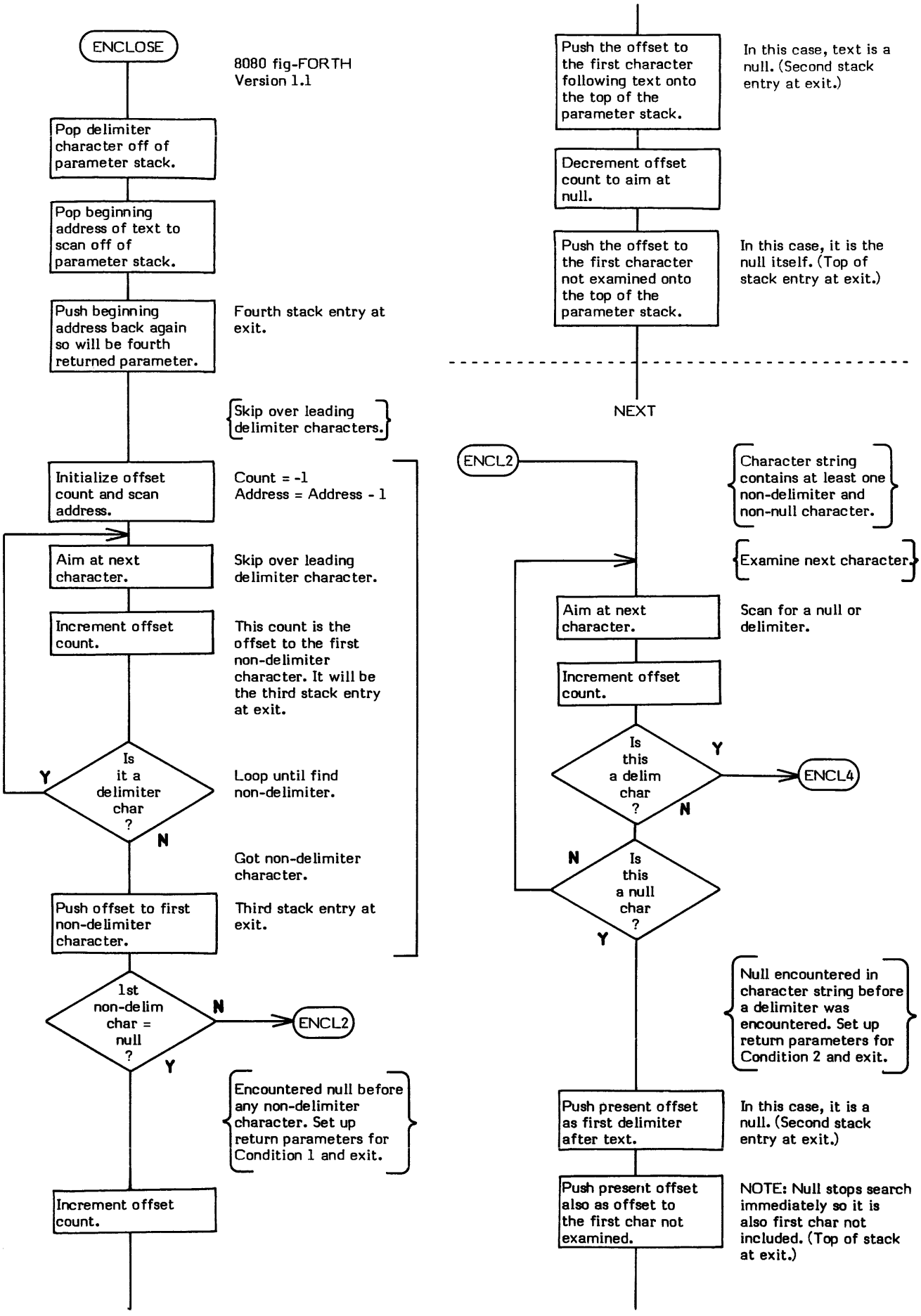
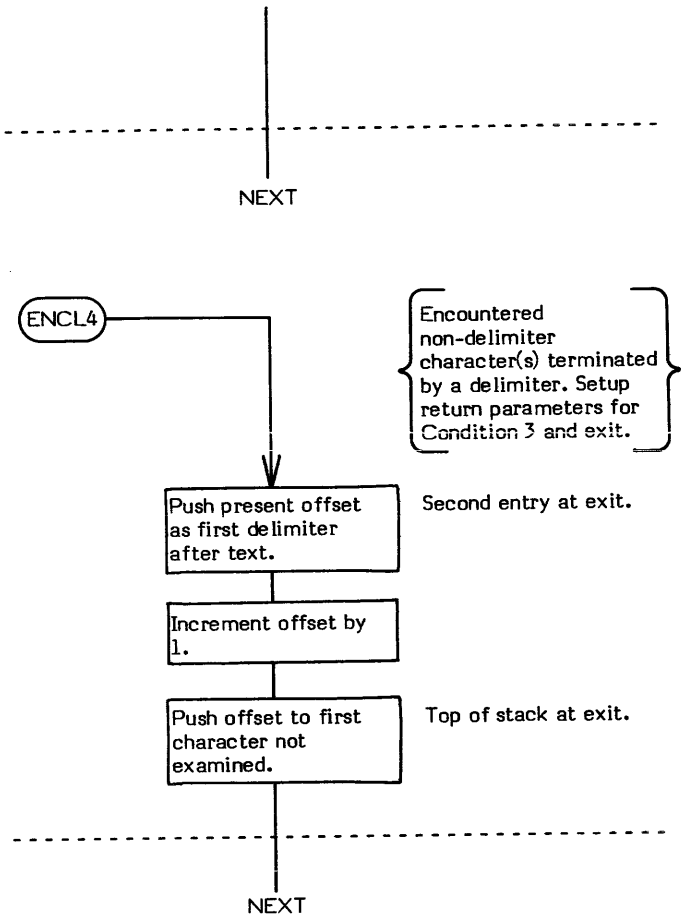


Figure ENCLOSE-1

High Level Flow Chart for ENCLOSE

NOTE: Each box roughly corresponds to a curly bracketed notation in the low level flow chart.


ENCLOSE
8080 fig-FORTH
Version 1.1
Pop delimiter character off of parameter stack.
Pop beginning address of text to scan off of parameter stack.
Push beginning address back again so will be fourth returned parameter.
Fourth stack entry at exit.
Skip over leading delimiter characters.
Initialize offset count and scan address.
Count = -1
Address = Address - 1
Aim at next character.
Skip over leading delimiter character.
Increment offset count.
This count is the offset to the first non-delimiter character. It will be the third stack entry at exit.
Is it a delimiter char ?
Y
N
Loop until find non-delimiter.
Got non-delimiter character.
Push offset to first non-delimiter character.
Third stack entry at exit.
1st non-delim char = null ?
N
Y
ENCL2
Encountered null before any non-delimiter character. Set up return parameters for Condition 1 and exit.
Increment offset count.
Push the offset to the first character following text onto the top of the parameter stack.
In this case, text is a null. (Second stack entry at exit.)
Decrement offset count to aim at null.
Push the offset to the first character not examined onto the top of the parameter stack.
In this case, it is the null itself. (Top of stack entry at exit.)
NEXT
ENCL2
Character string contains at least one non-delimiter and non-null character.
Examine next character.
Aim at next character.
Scan for a null or delimiter.
Increment offset count.
Is this a delim char ?
Y
N
ENCL4
Is this a null char ?
N
Y
Null encountered in character string before a delimiter was encountered. Set up return parameters for Condition 2 and exit.
Push present offset as first delimiter after text.
In this case, it is a null. (Second stack entry at exit.)
Push present offset also as offset to the first char not examined.
NOTE: Null stops search immediately so it is also first char not included. (Top of stack at exit.)


NEXT
ENCL4
Encountered non-delimiter character(s) terminated by a delimiter. Setup return parameters for Condition 3 and exit.
Push present offset as first delimiter after text.
Second entry at exit.
Increment offset by 1.
Push offset to first character not examined.
Top of stack at exit.
NEXT

END

END

COMPILE TIME (Sequence 2): (loop address\ 1 —)

EXECUTION TIME (Sequence 3): (truth flag —)

END is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

Note: "END" has been replaced by UNTIL . This is because END will eventually replace ; . UNTIL is the preferred usage, but END is still valid.

END is used to mark the end of an indefinite, conditional loop structure where repetition continues until the boolean input to END is true.

END is used in the form:

BEGIN "Loop Body" END

(UNTIL should be used in place of END .)

BEGIN-END structures must always be used within a colon definition.

It is important to note that in using a BEGIN-END structure, the "loop body" will always be executed at least once. This is because the exit condition is not tested "until" after the "loop body" has been executed. This is known as a "post-test" loop. If the exit condition must be tested before "loop body" execution, the BEGIN-WHILE-REPEAT structure should be used instead.

END is actually a Sequence 1 compiling word. That is END compiles (during Sequence 1) another compiling word, UNTIL , into its definition. Then (at Sequence 2) when END is executed, the word UNTIL does the actual compiling into the definition being created.

The apparent Sequence 2 compile time action of END is to compile a 0BRANCH into the dictionary. (Actually this is performed at Sequence 2 via the UNTIL which was compiled by END at Sequence 1.) Secondly, it resolves the "loop body" entry point address provided by BEGIN into a return branch offset used by the 0BRANCH and stores this offset into the definition.

Some compiler security is provided by checking for a 1 on the top of the stack. An END without a preceding BEGIN will probably not encounter a 1 on the top of the stack. (During compilation, BEGIN l saves a 1 on the top of the stack.)

The execution time action (Sequence 3) of END is to provide a conditional repetitive branch back to the loop's corresponding BEGIN (i.e., the beginning of the "loop body"). The input parameter to BEGIN is a boolean flag. If this flag is false (0), control returns to the first word in the "loop body" (just after BEGIN). If the boolean flag is true (not 0), no branch occurs and the loop is exited. That is, repetitive looping continues "until" the exit conditional is true.

The 0BRANCH, compiled into the definition at compile time, is what controls the looping.

Note that END is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 1 used for compiler security. The second stack entry contains the 16-bit entry point address of the "loop body" portion of the BEGIN-END structure.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains a 16-bit signed single precision boolean flag used to control the conditional looping of the structure.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) — This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) — There is some sort of problem with the pairing of conditionals within the definition being compiled.

END is a high level colon definition.

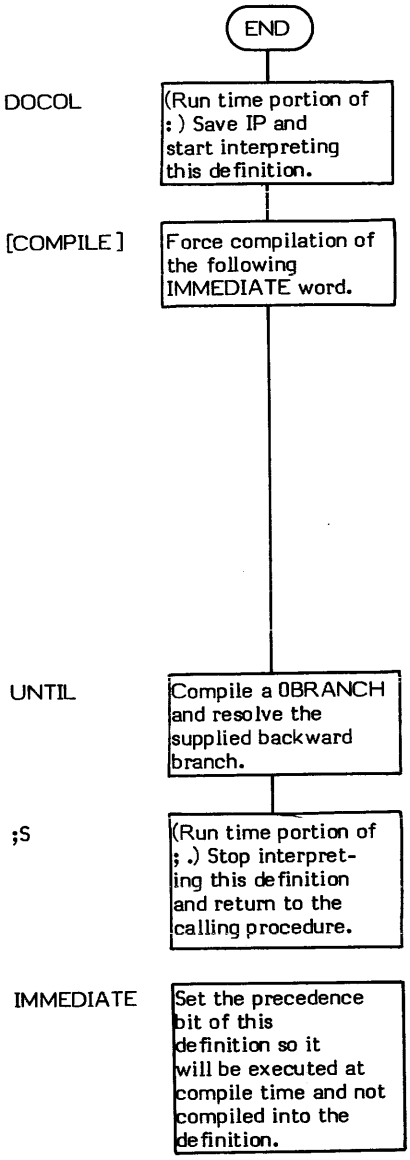
Refer to UNTIL , BEGIN and 0BRANCH .

FORTH-79: The FORTH-79 equivalent for END is UNTIL .

Definition:

```
:  END    ( loop address\ 1 -- )    ( compile time )
       [COMPILE] UNTIL    ;    IMMEDIATE
```

COMPILE TIME action of END (Sequence 1): (loop address \ 1 —)



END is simply a redefinition of UNTIL but since UNTIL is an IMMEDIATE word, [COMPILE] must be used to compile it into the definition. (Otherwise it would execute instead of being compiled.) [COMPILE] is only used (at Sequence 1) to compile UNTIL into END and does not exist in END 's definition when END is executed at compile time (at Sequence 2).

UNTIL will be executed at a later compile time (Sequence 2) when END is executed.

END is a compiler word and therefore must execute during compilation (Sequence 1) so that it can compile UNTIL into the definition.

EXECUTION TIME action of END (Sequence 3): (truth flag —)

Refer to UNTIL .

ENDIF

ENDIF

COMPILE TIME (Sequence 2): (offset address \ 2 —)

EXECUTION TIME (Sequence 3): (—)

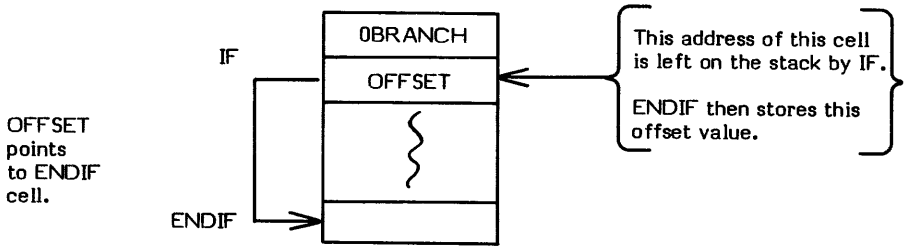
ENDIF is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

ENDIF is used to mark the end of an IF-ENDIF or IF-ELSE-ENDIF structure. (THEN is the preferred usage, but ENDIF is still valid.) ENDIF must be used in the form:

IF "true portion" ENDIF
OR
IF "true portion" ELSE "false portion" ENDIF

An IF-ENDIF structure must be used within a colon definition.

The compile time (Sequence 2) action of ENDIF is to compute a forward branch offset, calculated from the supplied Offset Address to the next available memory location following ENDIF (supplied by the HERE in ENDIF) and to store that offset in the location reserved by the previous IF or ELSE statement.



Some compiler security is provided by checking for a 2 on the top of the stack. IF and ELSE leave a 2 on the stack at compile time (Sequence 2). Since no other conditional or looping words leave a 2 on the stack, failure to pair an ENDIF with an IF or ELSE will cause an error condition to be detected and signaled via ?PAIRS .

The execution time (Sequence 3) action of ENDIF is to simply serve as the destination of a forward branch from a previous IF or ELSE statement. It is just a label.

Note that ENDIF is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 2 used for compiler security. The second stack entry contains a 16-bit address used to calculate the branch offset address of a previous IF or ELSE . The second stack entry is also the memory location that offset is to be stored into.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) — This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) — There is some sort of problem with the pairing of conditionals within the definition being compiled.

ENDIF is a high level colon definition.

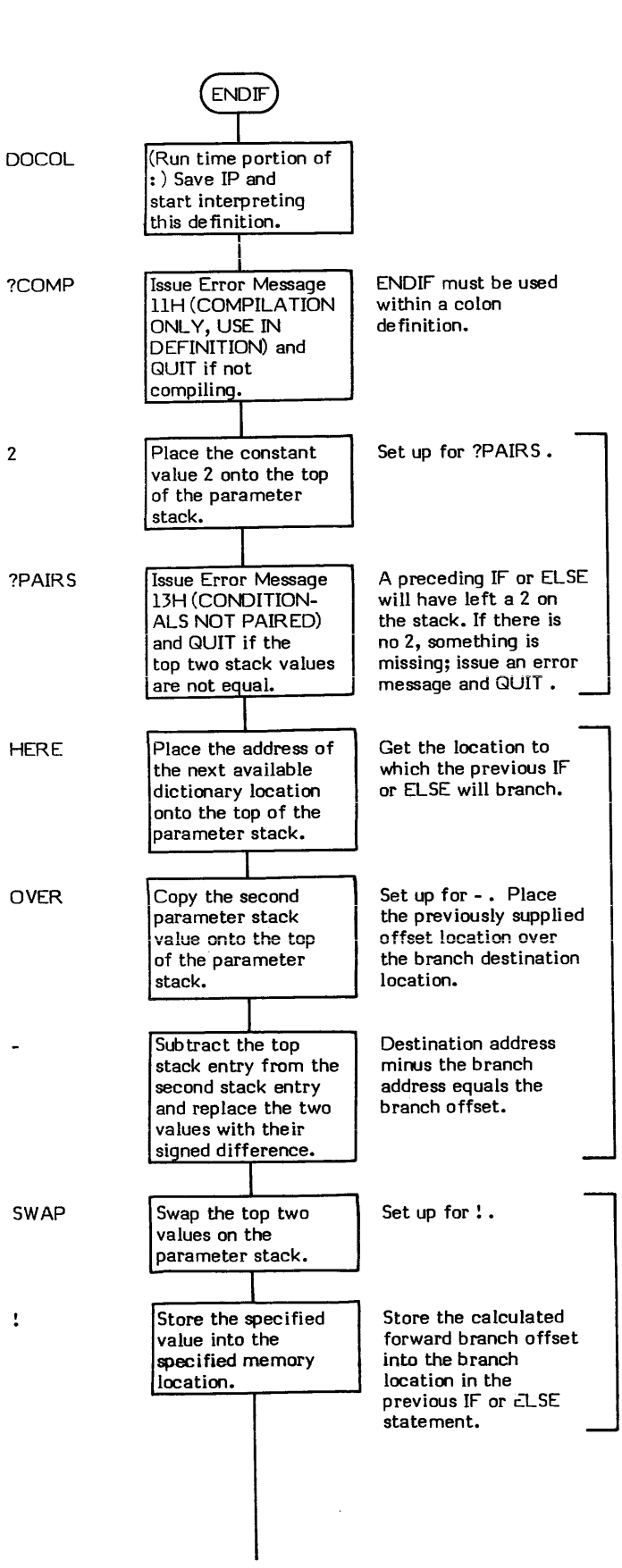
Refer to IF , ELSE , and THEN .

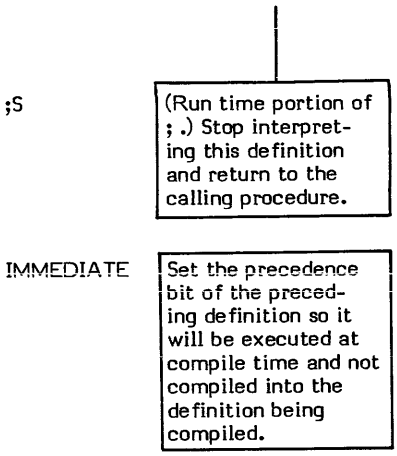
FORTH-79: The FORTH-79 equivalent for ENDIF is THEN .

Definition:

```
:  ENDIF  ( offset address \ 2 -- )  ( compile time )
       ?COMP  2 ?PAIRS  HERE OVER -  SWAP !  ;  IMMEDIATE
```

COMPILE TIME action of ENDIF (Sequence 2): (offset address \ 2 —)





EXECUTION TIME action of ENDIF (Sequence 3): (—)

ENDIF serves as the destination of a forward branch from a previous IF or ELSE statement. ENDIF is simply a label and has no run time code or action.

ERASE

ERASE (beginning address \ # of bytes to erase —)

ERASE clears a specified region of memory to zeros (0H).

ERASE is simply a FILL with the fill character (0) "hard coded".

EMPTY-BUFFERS is an example of word which uses ERASE.

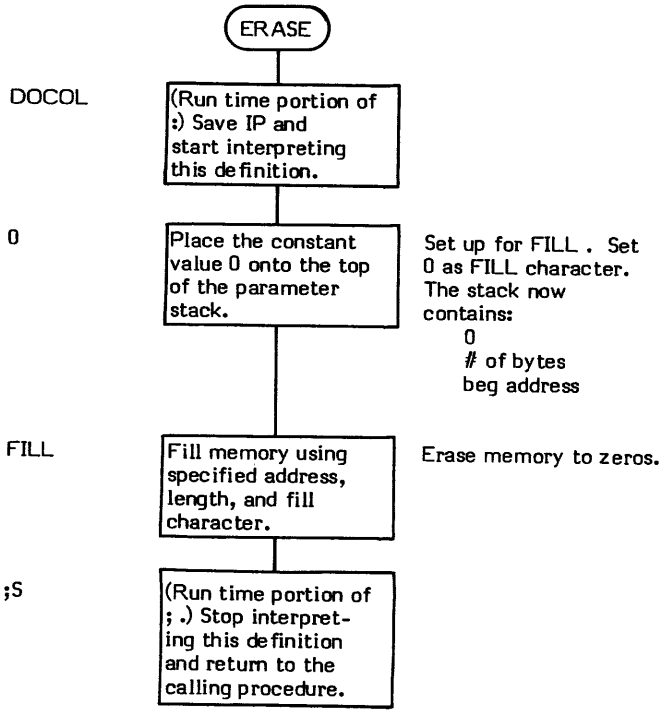
* **At entry** - The top of the parameter stack contains the 16-bit single precision number of bytes to erase. The second stack entry contains the 16-bit beginning address of the memory to erase.
* **At exit** - No parameters.

ERASE is a high level colon definition.

Refer to FILL .

FORTH-79: ERASE is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:
```
:  ERASE  ( beginning address\ # -- )
       0 FILL  ;
```



ERROR (**message # — contents of IN \ contents of BLK**)

ERROR displays some type of error message, stops compilation, and restarts interpretation from the terminal.

The form of the error message to be displayed is governed by the contents of the user variable WARNING .

If WARNING contains a negative number, a (ABORT) is performed and only the standard startup message is displayed. Note that (ABORT) may be modified to call a user created error routine.

If WARNING contains a zero, the message number supplied as an input parameter is simply printed as an "error number". This is often used for non-disk installations or when a disk that does not contain the error messages is being used (e.g., "MSG # 2").

If WARNING contains a non-zero positive number, the input parameter is used as a line number offset relative to Line 0 of Screen 4. The contents of this line is listed on the output device (e.g., "DICTIONARY FULL"). Note that message number 0 gives no message, just "?" (pronounced "huh?"). Also note that the error message number may be a negative value. It will still be referenced via Line 0 of Screen 4.

The contents of IN and BLK are saved to aid in determining where and why the error occurred.

Some systems use a word called WHERE which inputs this information and displays the line and word where the error occurred.

?ERROR is an example of a word which uses ERROR .

- **At entry** - The top of the parameter stack contains a signed 16-bit number used as an offset in lines (either positive or negative) relative to Line 0 of Screen 4. Either the number or the contents of the specified line will be displayed if WARNING contains a positive number.
- **At exit** - The top of the parameter stack contains the 16-bit block number of the block being interpreted when the error occurred (the contents of the user variable BLK). The second stack entry contains the 16-bit relative byte location of the next word to be interpreted (the contents of the user variable IN). These two parameters can then be used to aid in determining where and what the error was.

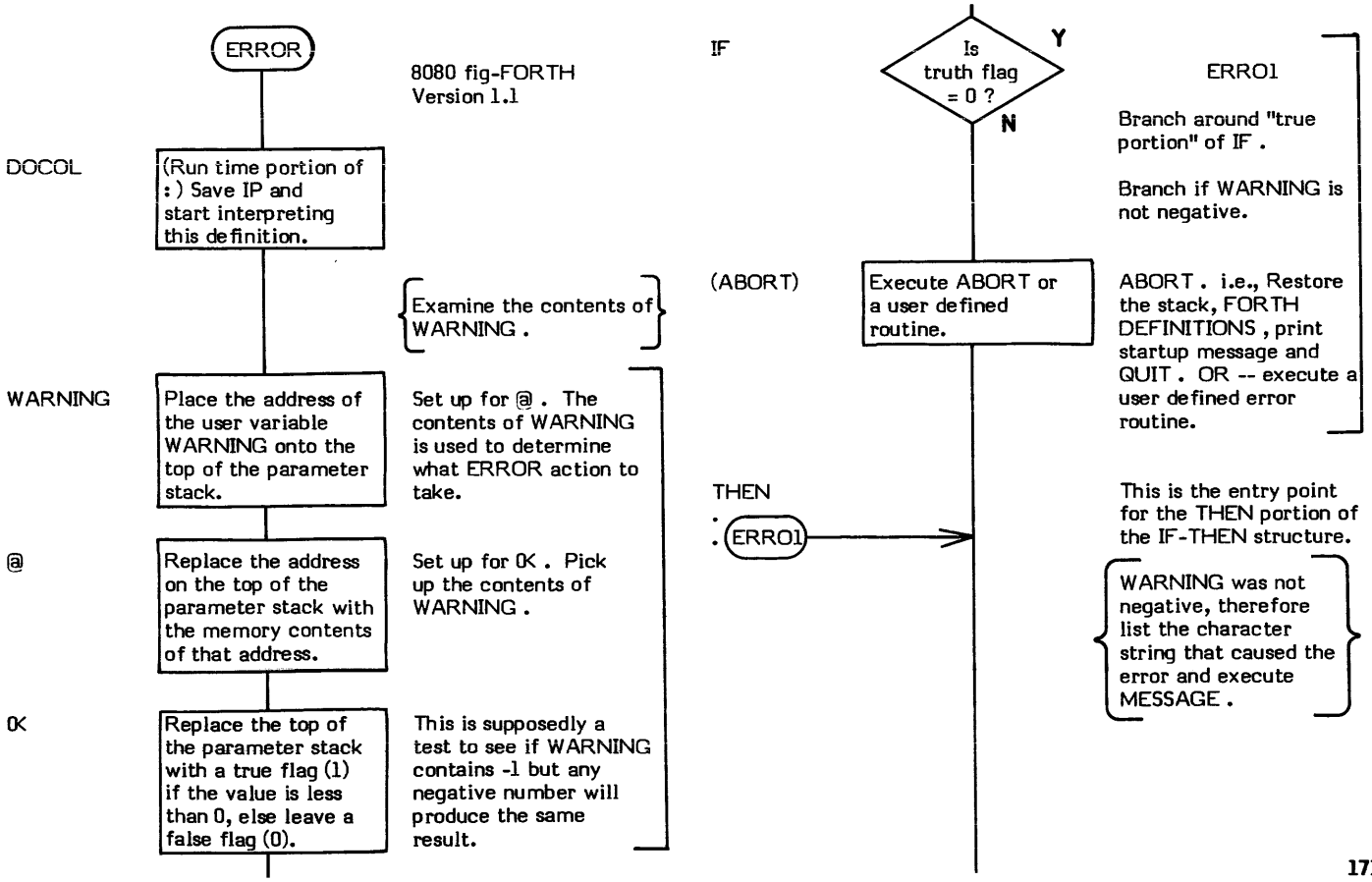
ERROR is a high level colon definition.

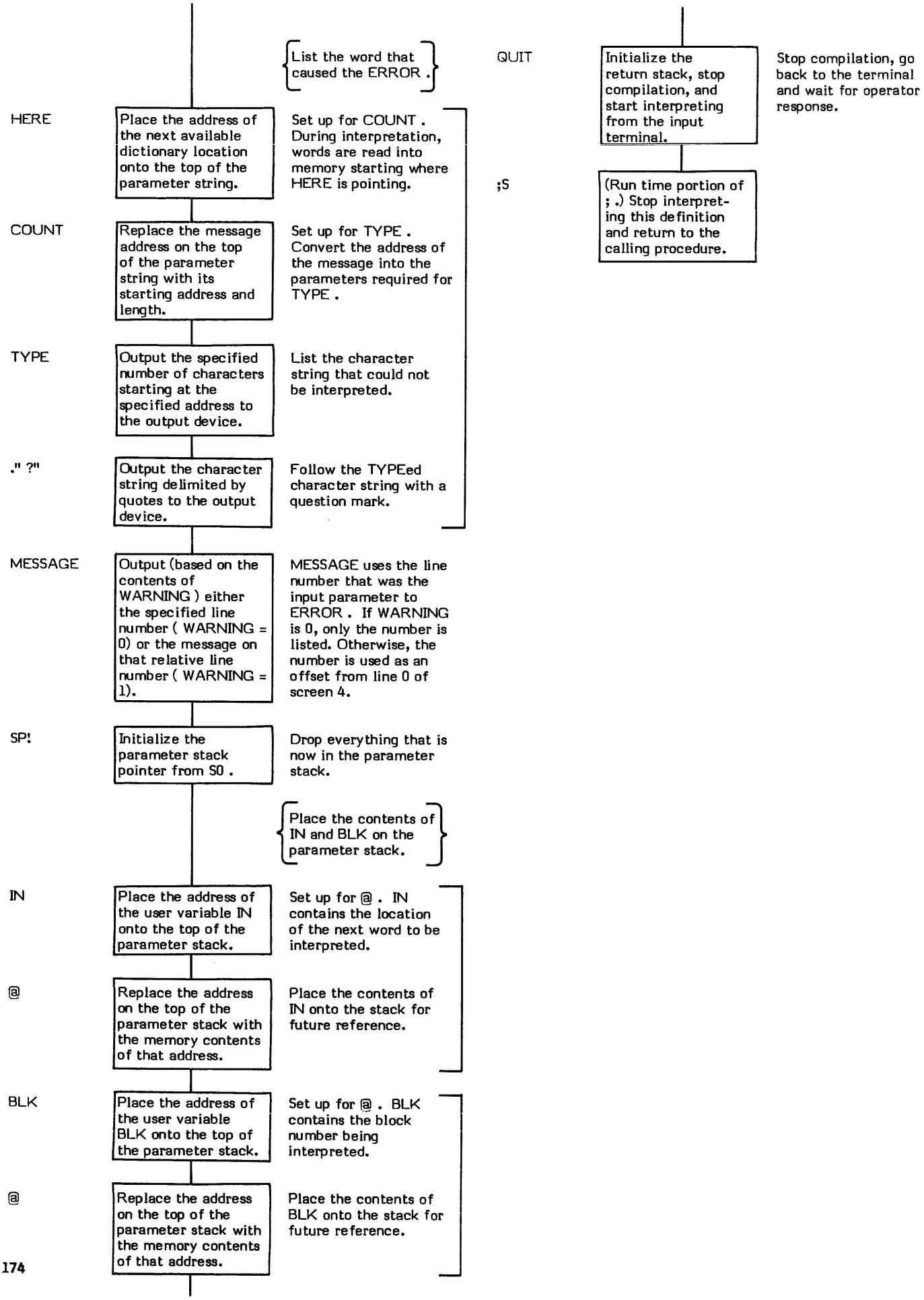
Refer to MESSAGE , WARNING , IN , and BLK .

FORTH-79: There is no FORTH-79 equivalent for ERROR .

Definition:

```
:  ERROR  ( message # -- IN\BLK )
     WARNING @ 0<  IF  (ABORT)  THEN
     HERE COUNT TYPE ."?"  MESSAGE  SP!
     IN @  BLK @  QUIT  ;
```




List the word that caused the ERROR .
HERE
Place the address of the next available dictionary location onto the top of the parameter string.
Set up for COUNT . During interpretation, words are read into memory starting where HERE is pointing.
COUNT
Replace the message address on the top of the parameter string with its starting address and length.
Set up for TYPE . Convert the address of the message into the parameters required for TYPE .
TYPE
Output the specified number of characters starting at the specified address to the output device.
List the character string that could not be interpreted.
." ?"
Output the character string delimited by quotes to the output device.
Follow the TYPEed character string with a question mark.
MESSAGE
Output (based on the contents of WARNING) either the specified line number (WARNING = 0) or the message on that relative line number (WARNING = 1).
MESSAGE uses the line number that was the input parameter to ERROR . If WARNING is 0, only the number is listed. Otherwise, the number is used as an offset from line 0 of screen 4.
SP!
Initialize the parameter stack pointer from S0 .
Drop everything that is now in the parameter stack.
Place the contents of IN and BLK on the parameter stack.
IN
Place the address of the user variable IN onto the top of the parameter stack.
Set up for @ . IN contains the location of the next word to be interpreted.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Place the contents of IN onto the stack for future reference.
BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for @ . BLK contains the block number being interpreted.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Place the contents of BLK onto the stack for future reference.
QUIT
Initialize the return stack, stop compilation, and start interpreting from the input terminal.
Stop compilation, go back to the terminal and wait for operator response.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

EXECUTE **(Code Field Address —)**

EXECUTE "executes" the definition whose Code Field Address (CFA) is on the top of the parameter stack. That is, EXECUTE transfers control to the definition whose CFA is on the top of the stack, executes that definition only, and then returns control to the next definition following EXECUTE .

This is action identical in philosophy to the IBM 360/370 EX (execute) instruction.

EXECUTE is the means by which vectored execution arrays can be used in FORTH.

EXECUTE is the "heart" of INTERPRET . The dictionary is searched for each incoming word in the data stream. When the word is found, its CFA is placed on the stack and the definition is then executed via EXECUTE .

The internal operation of EXECUTE is quite simple. The top of the stack is moved into W and then a jump indirect is performed through W. W is then incremented to point to the Code Field of the definition being executed.

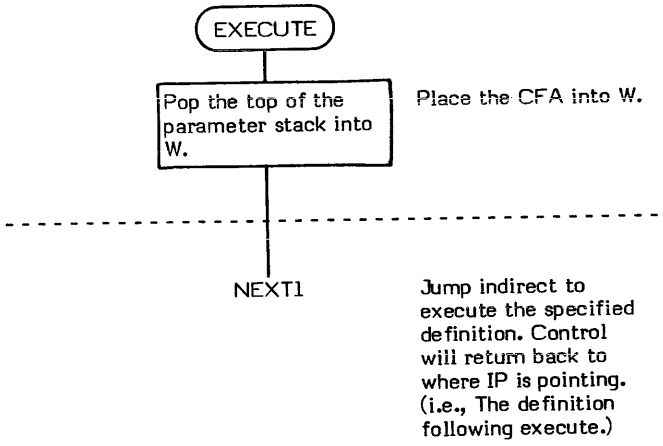
This operation is quite similar to that of NEXT except that the address of the next definition to execute comes from the top of the parameter stack instead of from IP.

* **At entry** - The top of the parameter stack contains the CFA of the definition to be "executed".
* **At exit** - No parameters.

EXECUTE is a low level code primitive.

Refer to NEXT , and INTERPRET .

FORTH-79: There is no FORTH-79 equivalent for EXECUTE .



EXPECT

EXPECT **(beginning text destination address \ character count —)**

EXPECT inputs characters from the terminal to the specified address until the specified count is reached or until a carriage return is encountered.

At least one null is appended to the end of the text stream. The carriage return character is always replaced with a blank followed by a null. The blank and null are delimiters for WORD and INTERPRET .

EXPECT checks for backspace characters. If a backspace is encountered, the loop Index is decremented so that the backspace is not included in the character count. The backspace character is still included in the text stream. If the backspace character was encountered in the first character position, i.e., trying to back up past the beginning; the character (which is an ascii 08) is decremented by 1 thereby converting it into an ascii BELL character (07). The BELL character is still included in the text stream. (The loop Index was decremented when the backspace was encountered.)

In multitasking FORTH environments, EXPECT is the primary task switching word. i.e., Input from any one of several terminals causes that terminal's task to become active.

QUERY is an example of a word which uses EXPECT .

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision character count value. This value is added to the specified address to determine the DO-LOOP Limit, so although negative values are legal, great care must be exercised in their use. The second stack entry contains the 16-bit beginning memory address where the text stream is to be stored.

* **At exit** - No parameters.

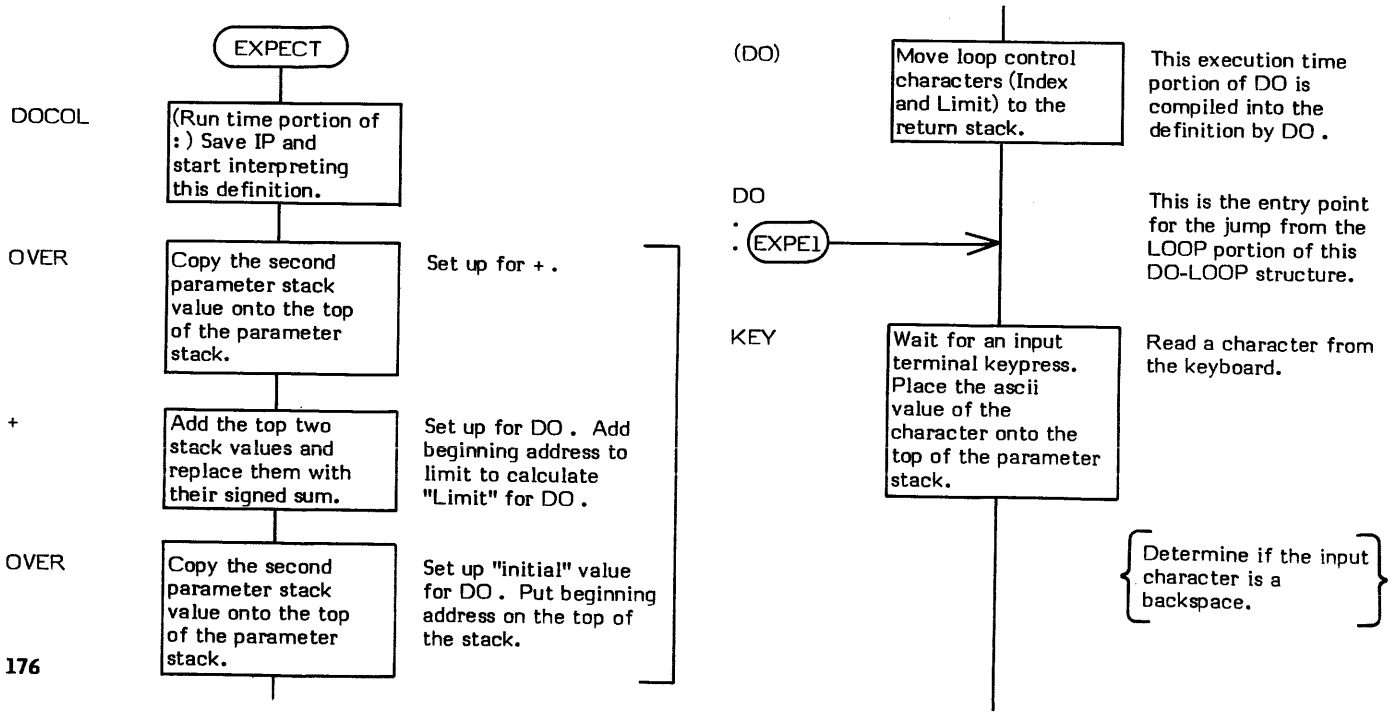
EXPECT is a high level colon definition.

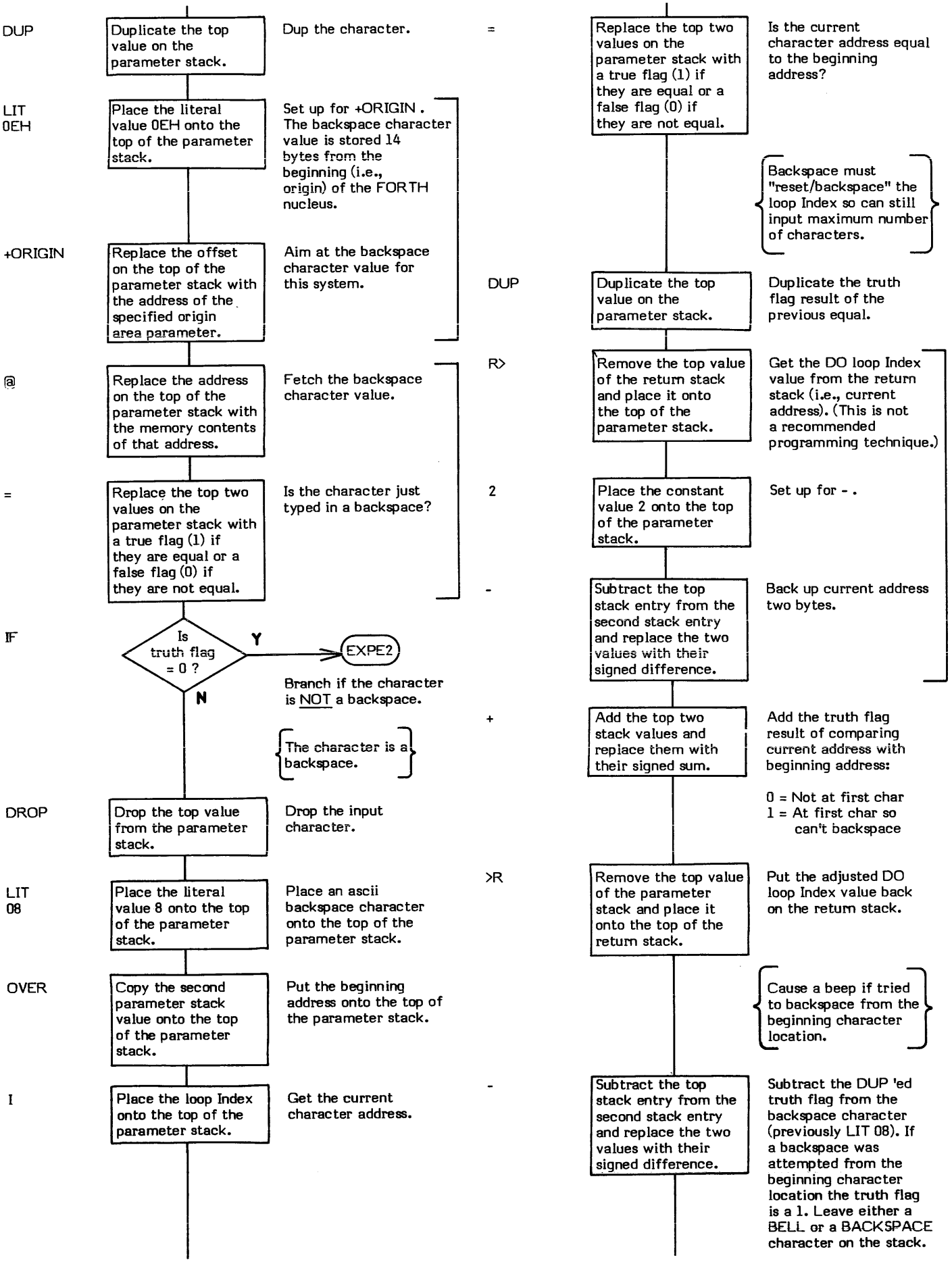
Refer to QUERY .

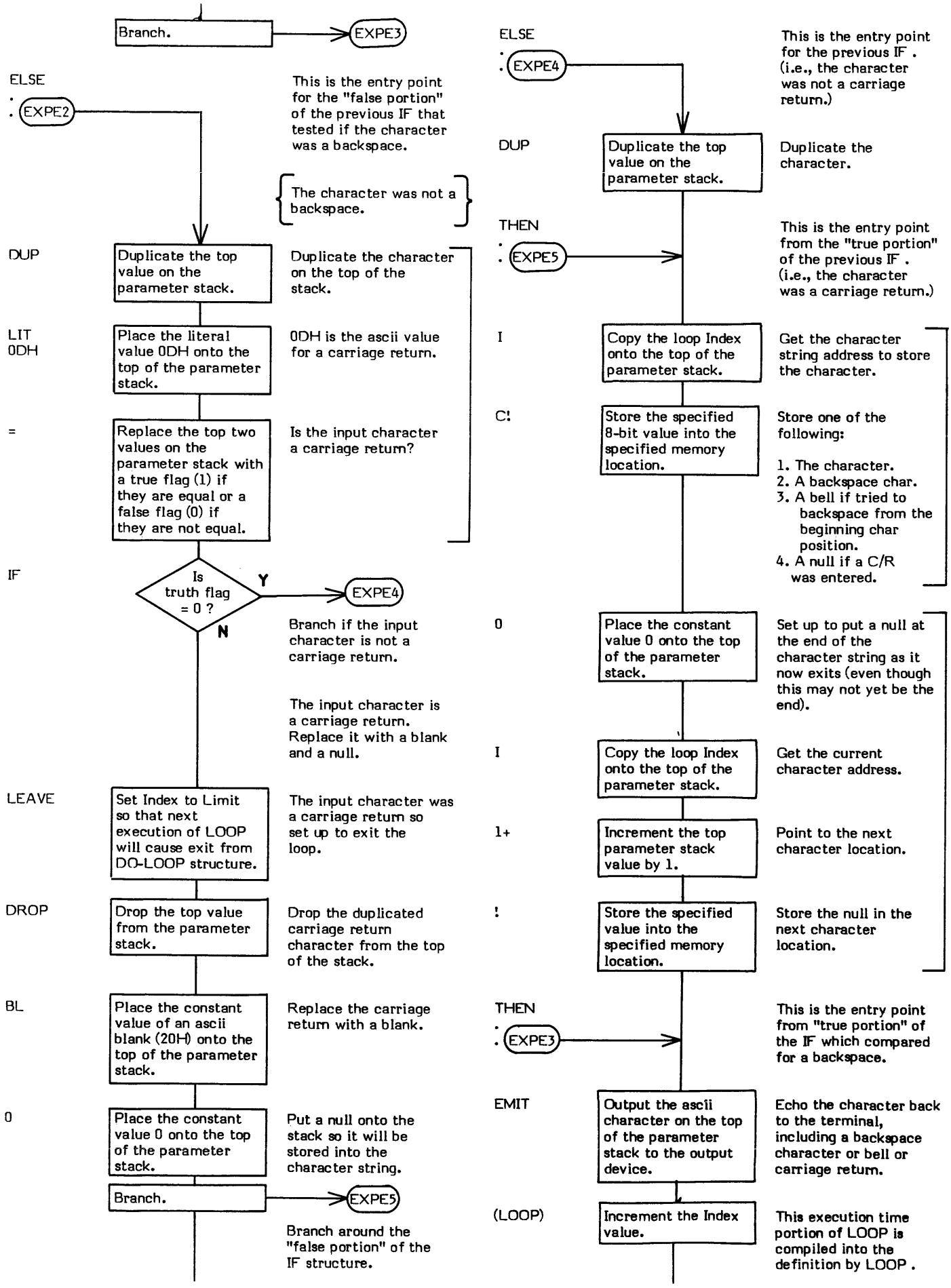
FORTH-79: The FORTH-79 equivalent for EXPECT is EXPECT .

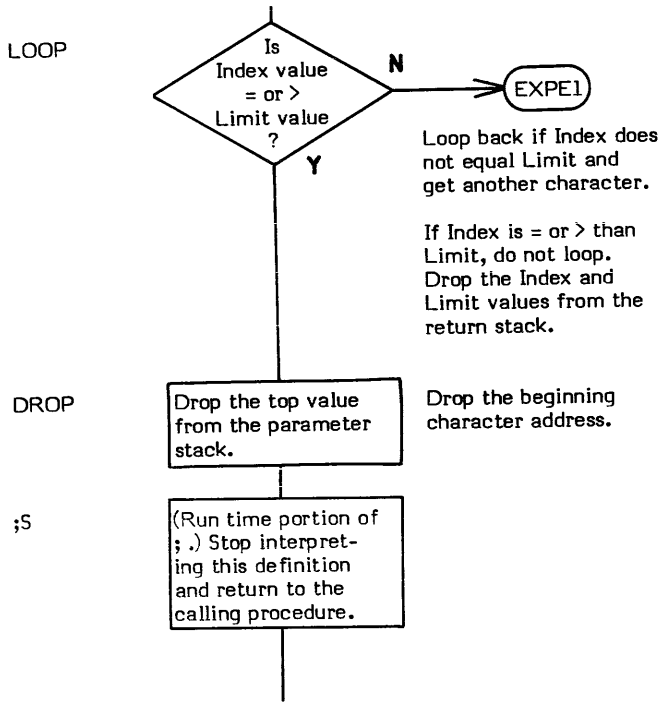
Definition:

```
:  EXPECT  ( beginning text destination address \ count -- )
     OVER + OVER
     DO
          KEY DUP   0E +ORIGIN   @ =
          IF
               DROP   08 OVER I = DUP   R> 2 -   + >R -
          ELSE
               DUP 0D =   IF
                               LEAVE   DROP BL 0
                          ELSE
                               DUP
                          THEN
               I C!   0 I 1+ !
          THEN
          EMIT
     LOOP   DROP   ;
```




DUP
Duplicate the top value on the parameter stack.
Dup the character.
LIT 0EH
Place the literal value 0EH onto the top of the parameter stack.
Set up for +ORIGIN . The backspace character value is stored 14 bytes from the beginning (i.e., origin) of the FORTH nucleus.
+ORIGIN
Replace the offset on the top of the parameter stack with the address of the specified origin area parameter.
Aim at the backspace character value for this system.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Fetch the backspace character value.
=
Replace the top two values on the parameter stack with a true flag (1) if they are equal or a false flag (0) if they are not equal.
Is the character just typed in a backspace?
IF
Is truth flag = 0 ?
Y
N
EXPE2
Branch if the character is NOT a backspace.
The character is a backspace.
DROP
Drop the top value from the parameter stack.
Drop the input character.
LIT 08
Place the literal value 8 onto the top of the parameter stack.
Place an ascii backspace character onto the top of the parameter stack.
OVER
Copy the second parameter stack value onto the top of the parameter stack.
Put the beginning address onto the top of the parameter stack.
I
Place the loop Index onto the top of the parameter stack.
Get the current character address.
=
Replace the top two values on the parameter stack with a true flag (1) if they are equal or a false flag (0) if they are not equal.
Is the current character address equal to the beginning address?
Backspace must "reset/backspace" the loop Index so can still input maximum number of characters.
DUP
Duplicate the top value on the parameter stack.
Duplicate the truth flag result of the previous equal.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Get the DO loop Index value from the return stack (i.e., current address). (This is not a recommended programming technique.)
2
Place the constant value 2 onto the top of the parameter stack.
Set up for - .
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Back up current address two bytes.
+
Add the top two stack values and replace them with their signed sum.
Add the truth flag result of comparing current address with beginning address:
0 = Not at first char
1 = At first char so can't backspace
>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Put the adjusted DO loop Index value back on the return stack.
Cause a beep if tried to backspace from the beginning character location.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Subtract the DUP 'ed truth flag from the backspace character (previously LIT 08). If a backspace was attempted from the beginning character location the truth flag is a 1. Leave either a BELL or a BACKSPACE character on the stack.


Branch.
EXPE3
ELSE
EXPE2
This is the entry point for the "false portion" of the previous IF that tested if the character was a backspace.
The character was not a backspace.
DUP
Duplicate the top value on the parameter stack.
Duplicate the character on the top of the stack.
LIT 0DH
Place the literal value 0DH onto the top of the parameter stack.
0DH is the ascii value for a carriage return.
=
Replace the top two values on the parameter stack with a true flag (1) if they are equal or a false flag (0) if they are not equal.
Is the input character a carriage return?
IF
Is truth flag = 0 ?
Y
N
EXPE4
Branch if the input character is not a carriage return.
The input character is a carriage return. Replace it with a blank and a null.
LEAVE
Set Index to Limit so that next execution of LOOP will cause exit from DO-LOOP structure.
The input character was a carriage return so set up to exit the loop.
DROP
Drop the top value from the parameter stack.
Drop the duplicated carriage return character from the top of the stack.
BL
Place the constant value of an ascii blank (20H) onto the top of the parameter stack.
Replace the carriage return with a blank.
0
Place the constant value 0 onto the top of the parameter stack.
Put a null onto the stack so it will be stored into the character string.
Branch.
EXPE5
Branch around the "false portion" of the IF structure.
ELSE
EXPE4
This is the entry point for the previous IF . (i.e., the character was not a carriage return.)
DUP
Duplicate the top value on the parameter stack.
Duplicate the character.
THEN
EXPE5
This is the entry point from the "true portion" of the previous IF . (i.e., the character was a carriage return.)
I
Copy the loop Index onto the top of the parameter stack.
Get the character string address to store the character.
C!
Store the specified 8-bit value into the specified memory location.
Store one of the following:
1. The character.
2. A backspace char.
3. A bell if tried to backspace from the beginning char position.
4. A null if a C/R was entered.
0
Place the constant value 0 onto the top of the parameter stack.
Set up to put a null at the end of the character string as it now exits (even though this may not yet be the end).
I
Copy the loop Index onto the top of the parameter stack.
Get the current character address.
1+
Increment the top parameter stack value by 1.
Point to the next character location.
!
Store the specified value into the specified memory location.
Store the null in the next character location.
THEN
EXPE3
This is the entry point from "true portion" of the IF which compared for a backspace.
EMIT
Output the ascii character on the top of the parameter stack to the output device.
Echo the character back to the terminal, including a backspace character or bell or carriage return.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .


LOOP
Is
Index value
= or >
Limit value
?
N
Y
EXPE1
Loop back if Index does not equal Limit and get another character.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
DROP
Drop the top value from the parameter stack.
Drop the beginning character address.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

FENCE

FENCE (— data address)

FENCE is a user variable used in conjunction with FORGET to set a boundary past which FORGET cannot forget. This prevents the programmer from inadvertently forgetting too much of the dictionary. FENCE is initialized at start up time by COLD with data from the origin parameter area but its contents may be changed at any time to reflect a new dictionary structure.

The words:

```
'FROG FENCE !
```

will not allow forgetting below the named definition (in this case, FROG).

The user variable FENCE is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable FENCE .

Refer to FORGET , and USER .

FORTH-79: There is no FORTH-79 equivalent for FENCE .

FILL (start address \ count \ fill character —)

FILL "fills" each location of a specified region of memory with a specified byte value.

Note: This word may be installation dependent on other than 8-bit machines.

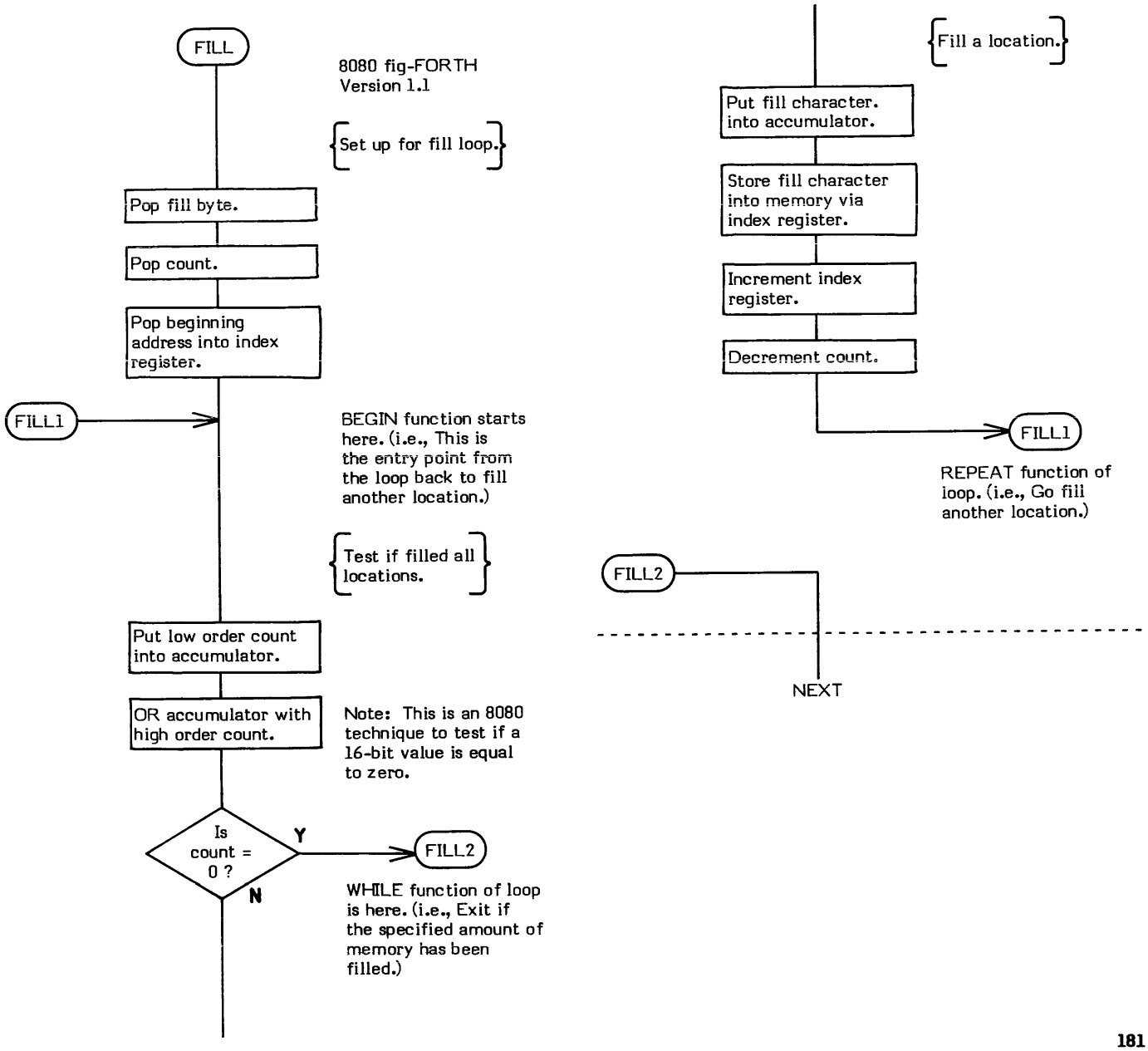
This version of FILL differs from the fig-FORTH model in that it is a low level code primitive. The high level version of FILL (based on CMOVE) works correctly when filling read/write memory. However, it will not work correctly when filling write-only memory such as specialized video display memory. This is because the "ripple fill" used by CMOVE requires that the byte to be moved be read from memory. This code version of FILL does not use CMOVE or "ripple fill" and therefore will work correctly when filling write-only memory.

ERASE and BLANKS are examples of words that use FILL .

* **At entry** - The top of the parameter stack contains a 16-bit value of which the low order 8-bits are the fill character. The second stack entry contains a 16-bit unsigned "count", i.e., the number of locations to fill. The third stack entry contains a 16-bit "starting address" (inclusive).
* **At exit** - No parameters.

FILL is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for FILL is FILL .



FIRST

FIRST (— address)

FIRST is a single precision CONSTANT value. This constant places the memory address of the "first" (lowest) block buffer in the buffer array onto the top of the parameter stack. Refer to +BUF for an example of the usage of FIRST . Also see BLOCK and BUFFER for a description of the buffer array.

Conversely, LIMIT is the constant that reflects the upper "limit" (highest) block buffer address.

* **At entry -** No parameters.
* **At exit -** The top of the parameter stack contains the 16-bit memory address of the first buffer in the buffer-array.

Refer to LIMIT , +BUF , BLOCK , and BUFFER .

FORTH-79: There is no FORTH-79 equivalent for FIRST .

FLD (-- data address)

FLD (pronounced "F-L-D") is a user variable which is intended to control field length in pictured numeric output.

FLD is not currently used in the fig-FORTH model.

The user variable FLD is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable FLD .

FORTH-79: There is no FORTH-79 equivalent for FLD .

FLUSH

FLUSH (–)

FLUSH writes to disk all the buffers in the buffer array that have been flagged as "updated".

FLUSH is FORTH's virtual memory equivalent to "traditional" languages "write-to-disk" command.

A buffer is flagged as being "updated" when the most significant bit (i.e., the update bit) in the header portion of a buffer is set to one. (Refer to +BUF for a detailed explanation of the buffer structure.) This update bit is set via the word UPDATE .

Flagged buffers will automatically be written to disk by BUFFER when the buffer is allocated to a new block number while executing BLOCK . Sometimes it is desirable or even necessary to force writing of these updated buffers from memory to disk before this re-allocation takes place.

Examples of when it is desirable to FLUSH the buffers are:

1. Before changing disks (this ensures that the updated data is written to the proper disk).
2. Before leaving FORTH.
3. Before powering down the system.

Typing FLUSH immediately after finishing editing a source screen is a good habit to develop to prevent inadvertently removing a disk before the newly edited data is written to it.

If there are no "updated" buffers (buffers with a set update bit), FLUSH has no effect. Executing EMPTY-BUFFERS prior to executing FLUSH means no data will be written to disk.

The basis of FLUSH is BUFFER . The word BUFFER is contained within a DO-LOOP which is executed as many times as there are buffers in the buffer array. BUFFER then re-allocates each buffer and writes any "updated" buffer data to disk.

Note that, within FLUSH , BUFFER re-allocates each buffer to block number 0 (8080 fig-FORTH Version 1.1). A side effect of this operation is that a reference (with BLOCK) to a block that was in memory prior to the execution of FLUSH will cause a disk access. This happens because the buffer allocation scheme has been re-initialized even though the data portion of the block is still in memory.

* **At entry** - No parameters.
* **At exit** - No parameters.

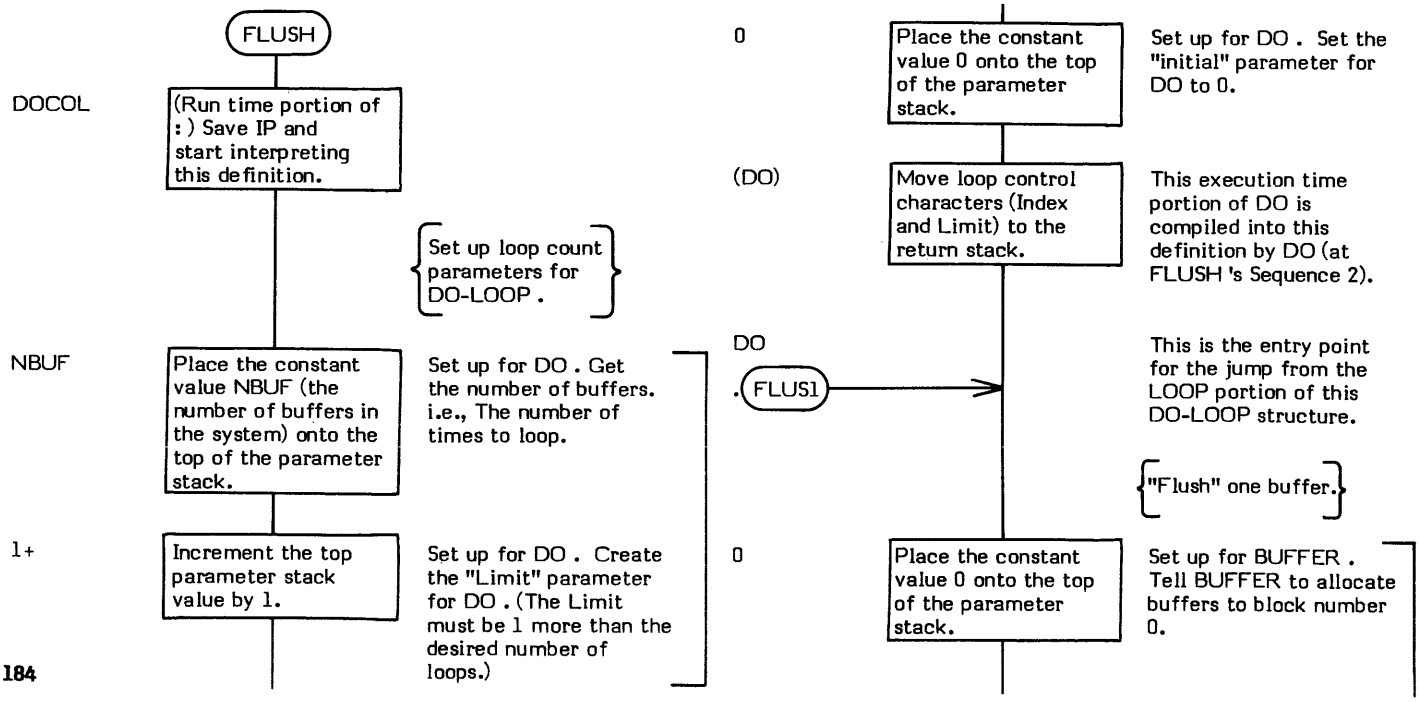
FLUSH is a high level colon definition.

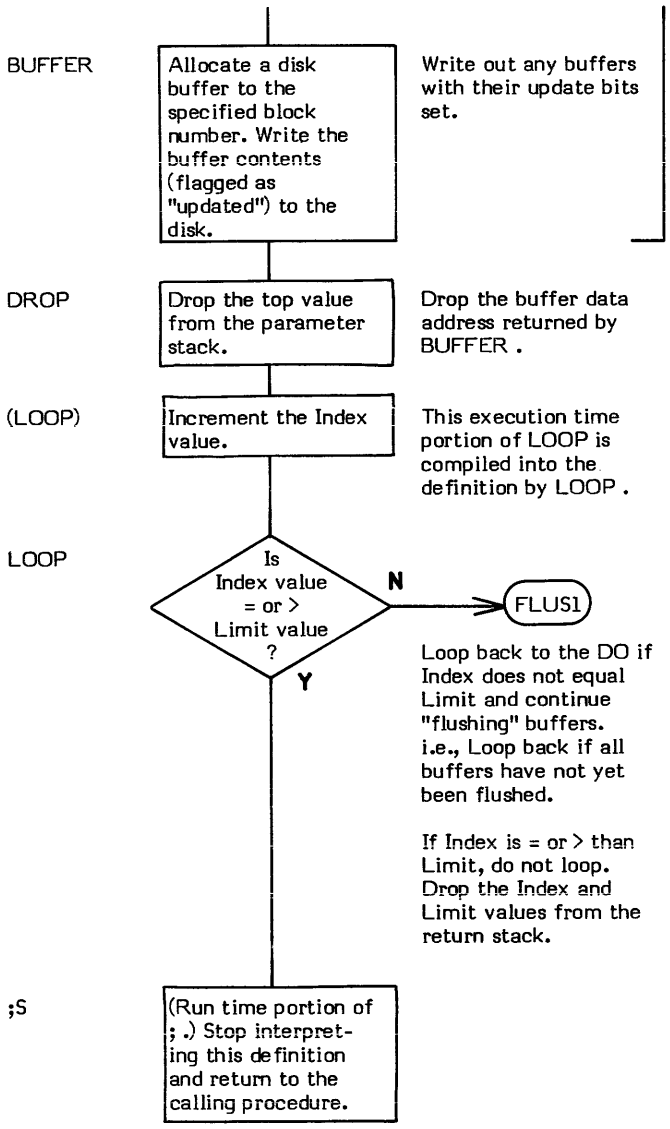
Refer to BUFFER , +BUF , UPDATE , and BLOCK .

FORTH-79: The FORTH-79 equivalent for FLUSH is SAVE-BUFFERS .

Definition:

```
: FLUSH ( – )
    NBUF 1+  0 DO
                 0 BUFFER  DROP
             LOOP ;
```




BUFFER
Allocate a disk buffer to the specified block number. Write the buffer contents (flagged as "updated") to the disk.
Write out any buffers with their update bits set.
DROP
Drop the top value from the parameter stack.
Drop the buffer data address returned by BUFFER .
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
FLUSl
Y
Loop back to the DO if Index does not equal Limit and continue "flushing" buffers. i.e., Loop back if all buffers have not yet been flushed.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

FORGET

FORGET (–)

FORGET deletes definitions from the dictionary. The specified definition and all definitions following, up until the end of the dictionary, are "forgotten".

FORGET is used in the form:

FORGET "definition name"

In the fig-FORTH version of FORGET , the programmer must ensure that the CONTEXT and CURRENT vocabularies are the same or the error message DECLARE VOCABULARY will be issued and the FORGET will QUIT or ABORT .

FORGET searches the dictionary for the first occurrence of the specified definition name and changes DP (the Dictionary Pointer) to point to the located definition. That definition (and all following) is "forgotten" and will be overlayed when the next new definition is created. The CONTEXT and CURRENT vocabulary linkages are also modified to drop the forgotten definitions from the vocabulary chain.

FORGET is a very powerful word. It is possible to forget the entire dictionary, including FORTH itself. To limit the scope of this powerful word and to prevent accidental forgetting of important definitions, the user variable FENCE is used. FENCE is a user variable which contains an address below which it is not possible to FORGET .

WARNING: Using FORGET in a system comprised of multiple vocabularies will probably (but not necessarily always) cause the system to crash.

FORGET , as defined in the fig-FORTH Model, is "unaware" of any vocabularies other than the vocabulary referred to by CONTEXT and CURRENT . As described in VOCABULARY , multiple vocabularies are logically structured like branches on a tree; but they are physically structured as a one-dimensional linked list. The fig-FORTH Model version of FORGET simply uses "brute force" to forget a portion of this one-dimensional dictionary from the end up to and including the specified definition. The fact that there may be vocabulary links which now point into a "forgotten" dictionary is not taken into account. Any reference to such a vocabulary having a "broken" chain will produce undeterminable results.

* **At entry** - No parameter stack entries. The word immediately following FORGET in the input data stream is the name of the definition to be "forgotten".
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

PROTECTED DICTIONARY (15H) -- The address of the definition being "forgotten" is less than the value stored in FENCE . Change the value in FENCE .

DECLARE VOCABULARY (18H) -- CONTEXT and CURRENT are not aiming at the same vocabularies when attempting to FORGET .

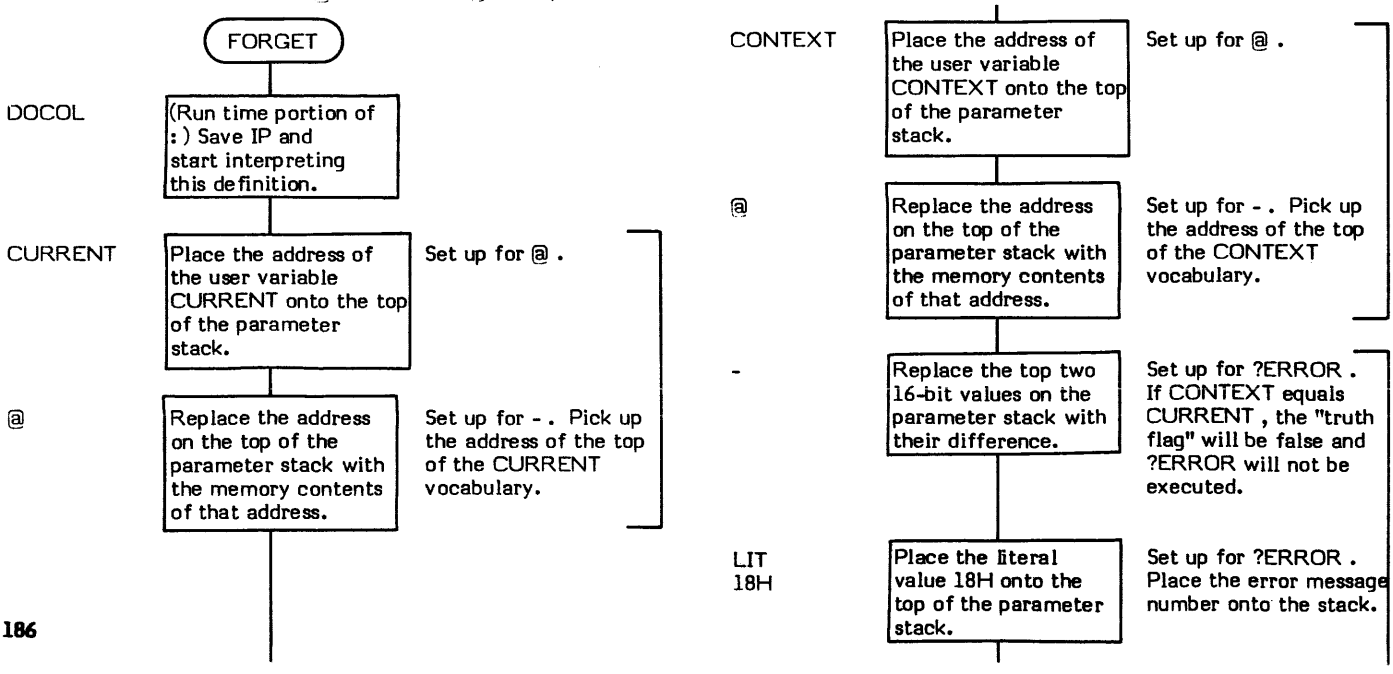
FORGET is a high level colon definition.

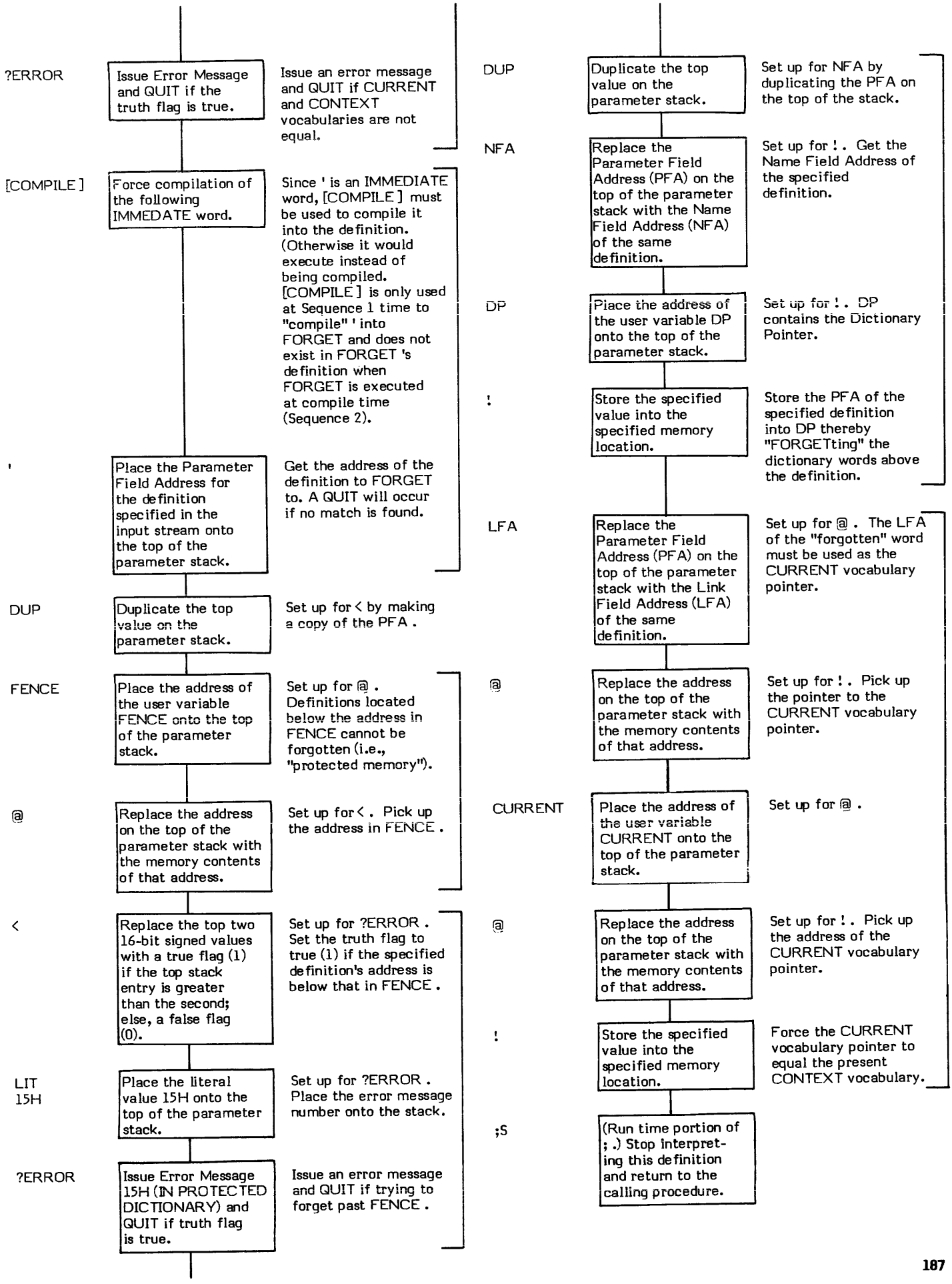
Refer to FENCE , and VOCABULARY .

FORTH-79: The FORTH-79 equivalent for FORGET is FORGET .

Definition:

```
: FORGET ( – )
    CURRENT @  CONTEXT @  - 18 ?ERROR  [COMPILE ] '
    DUP  FENCE @  < 15 ?ERROR  DUP NFA DP !
    LFA @ CURRENT @ !  ;
```




?ERROR
Issue Error Message and QUIT if the truth flag is true.
Issue an error message and QUIT if CURRENT and CONTEXT vocabularies are not equal.
[COMPILE]
Force compilation of the following IMMEDATE word.
Since ' is an IMMEDIATE word, [COMPILE] must be used to compile it into the definition. (Otherwise it would execute instead of being compiled. [COMPILE] is only used at Sequence 1 time to "compile" ' into FORGET and does not exist in FORGET 's definition when FORGET is executed at compile time (Sequence 2).
'
Place the Parameter Field Address for the definition specified in the input stream onto the top of the parameter stack.
Get the address of the definition to FORGET to. A QUIT will occur if no match is found.
DUP
Duplicate the top value on the parameter stack.
Set up for < by making a copy of the PFA .
FENCE
Place the address of the user variable FENCE onto the top of the parameter stack.
Set up for @ . Definitions located below the address in FENCE cannot be forgotten (i.e., "protected memory").
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for < . Pick up the address in FENCE .
<
Replace the top two 16-bit signed values with a true flag (1) if the top stack entry is greater than the second; else, a false flag (0).
Set up for ?ERROR . Set the truth flag to true (1) if the specified definition's address is below that in FENCE .
LIT
15H
Place the literal value 15H onto the top of the parameter stack.
Set up for ?ERROR . Place the error message number onto the stack.
?ERROR
Issue Error Message 15H (IN PROTECTED DICTIONARY) and QUIT if truth flag is true.
Issue an error message and QUIT if trying to forget past FENCE .
DUP
Duplicate the top value on the parameter stack.
Set up for NFA by duplicating the PFA on the top of the stack.
NFA
Replace the Parameter Field Address (PFA) on the top of the parameter stack with the Name Field Address (NFA) of the same definition.
Set up for ! . Get the Name Field Address of the specified definition.
DP
Place the address of the user variable DP onto the top of the parameter stack.
Set up for ! . DP contains the Dictionary Pointer.
!
Store the specified value into the specified memory location.
Store the PFA of the specified definition into DP thereby "FORGETting" the dictionary words above the definition.
LFA
Replace the Parameter Field Address (PFA) on the top of the parameter stack with the Link Field Address (LFA) of the same definition.
Set up for @ . The LFA of the "forgotten" word must be used as the CURRENT vocabulary pointer.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for ! . Pick up the pointer to the CURRENT vocabulary pointer.
CURRENT
Place the address of the user variable CURRENT onto the top of the parameter stack.
Set up for @ .
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for ! . Pick up the address of the CURRENT vocabulary pointer.
!
Store the specified value into the specified memory location.
Force the CURRENT vocabulary pointer to equal the present CONTEXT vocabulary.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

FORTH

FORTH (–)

FORTH is the name of the "vocabulary" definition for the FORTH vocabulary. The FORTH vocabulary contains all of the standard FORTH definitions. Executing FORTH sets CONTEXT to point to the FORTH vocabulary. This will cause dictionary searches to begin with the FORTH vocabulary. Since CONTEXT now points to FORTH , executing DEFINITIONS would then cause new definitions to be appended to the FORTH vocabulary.

The FORTH vocabulary is the topmost vocabulary. All other vocabularies will be appended under FORTH . At cold start initialization, the user variable VOC-LINK is initialized to point to FORTH .

The FORTH definition is created via the word VOCABULARY . VOCABULARY is a <BUILDS DOES> definition. This means that the <BUILDS portion creates the actual FORTH definition in the dictionary at Sequence 2 time. The DOES> portion of the VOCABULARY definition contains the definitions that will be executed when FORTH is executed at Sequence 3 time. i.e., The FORTH definition contains a pointer to its execution time procedure which "lives" in VOCABULARY and at execution time the run time code for DOES> transfers control to that procedure in VOCABULARY .

A more detailed explanation of vocabularies can be found in VOCABULARY . The definition FORTH is used as an example in the descriptions of the words <BUILDS and DOES> .

* **At entry** - No parameters.
* **At exit** - No parameters.

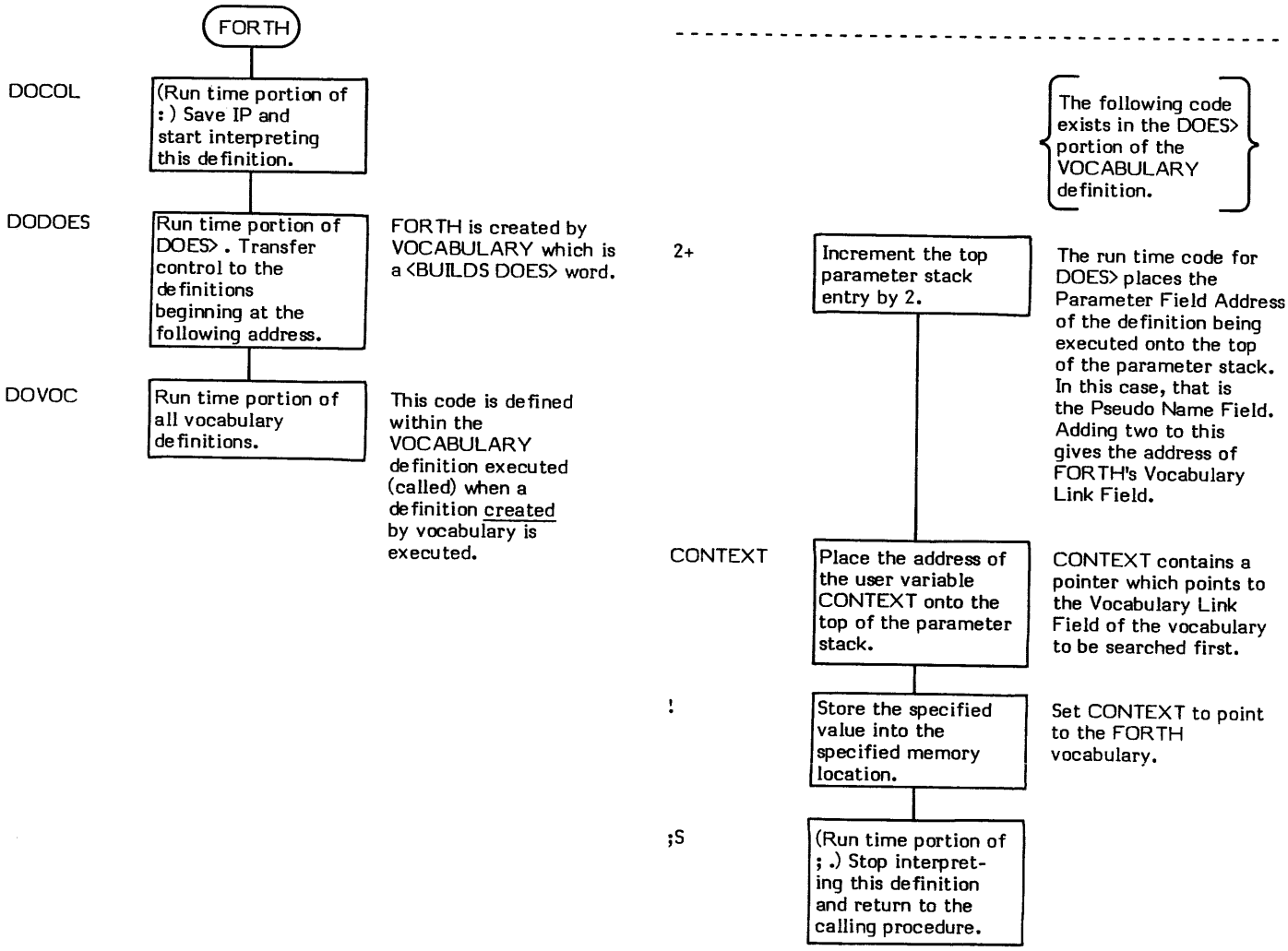
FORTH is a high level colon definition.

Refer to VOCABULARY , CONTEXT , CURRENT , DEFINITIONS , <BUILDS , DOES> , and VOC-LINK .

FORTH-79: The FORTH-79 equivalent for FORTH is FORTH .

Definition:

```
:  FORTH  ( -- )
     VOCABULARY  FORTH  ;  IMMEDIATE
```



HERE **(-- address)**

HERE places the address of the next available dictionary location onto the parameter stack. The use of HERE is the proper way to obtain the location of the end of the dictionary (e.g., DP @ on some implementations will return a wrong address).

HERE is used in such basic words as , (comma) and also in higher-level words such as ." (dot-quote).

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the next available dictionary location.

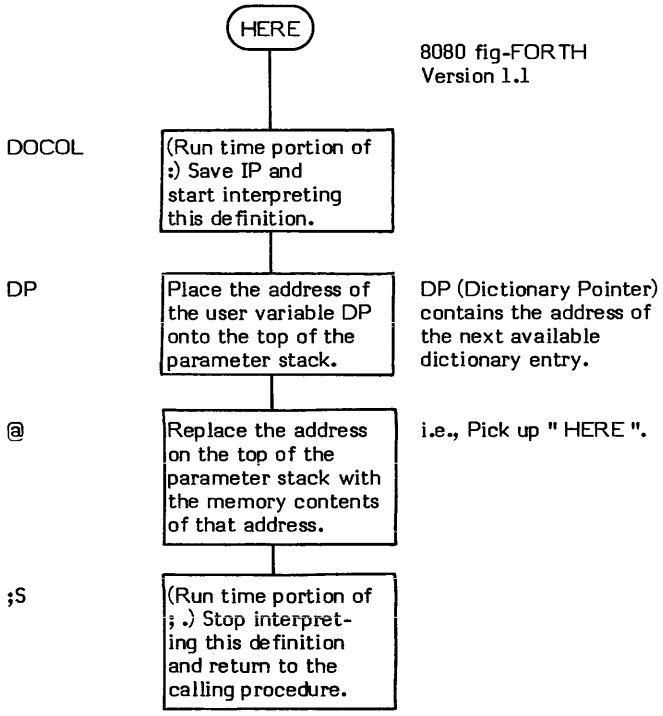
HERE is a high level colon definition.

Refer to DP .

FORTH-79: The FORTH-79 equivalent for HERE is HERE .

Definition:

```
: HERE ( -- address )
    DP @ ;
```



HEX

HEX (–)

HEX sets the user variable BASE to 16 (decimal). This causes all numeric input and output conversions to be performed in hexidecimal (base 16).

Note that this is not an IMMEDIATE word.

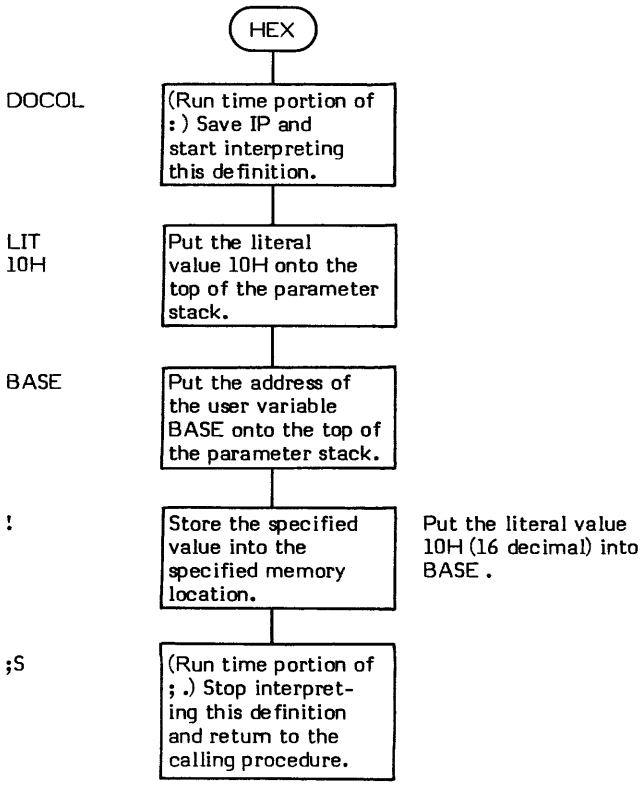
* **At entry** - No parameters.
* **At exit** - No parameters.

HEX is a high level colon definition.

Refer to (NUMBER) , and BASE .

FORTH-79: The FORTH-79 equivalent for HEX is HEX .

Definition:
```
: HEX ( – )
    10 BASE ! ;
```



HLD (– data address)

HLD (pronounced "H-L-D") is a user variable which contains the address of the last character of text placed into PAD during binary-to-ascii pictured numeric output conversion. <# initializes HLD and HOLD references HLD during conversion.

The user variable HLD is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable HLD .

Refer to <# , #> , HOLD , SIGN , NUMBER , (NUMBER) , and USER .

FORTH-79: There is no FORTH-79 equivalent for HLD .

HOLD

HOLD (ascii char –)

HOLD stores an ascii character into the next available location in a pictured numeric output string. HOLD is used internally by # to place each converted ascii digit into memory. Numeric punctuation (characters such as commas, decimal points, signs, dollar signs, etc.) can also be placed in output strings via HOLD .

The address of the next available location is kept in the user variable HLD . HOLD decrements HLD before each character store. HOLD is used within a <# #> expression. <# is used to initialize HLD to the beginning of PAD . Therefore characters are stored into memory from high to low memory starting from one byte before the beginning of PAD working towards the end of the dictionary.

SIGN uses HOLD to place a minus sign into an output string. The description of <# and # explain the operation of pictured numeric output.

* **At entry** - The top of the parameter stack contains an 8-bit value (normally an ascii character) in the low order portion of the topmost 16-bit word. The user variable HLD contains the memory location +1 where the character is to be stored.
* **At exit** - No parameters. However, the character is in memory and the contents of HLD has been decremented by 1.

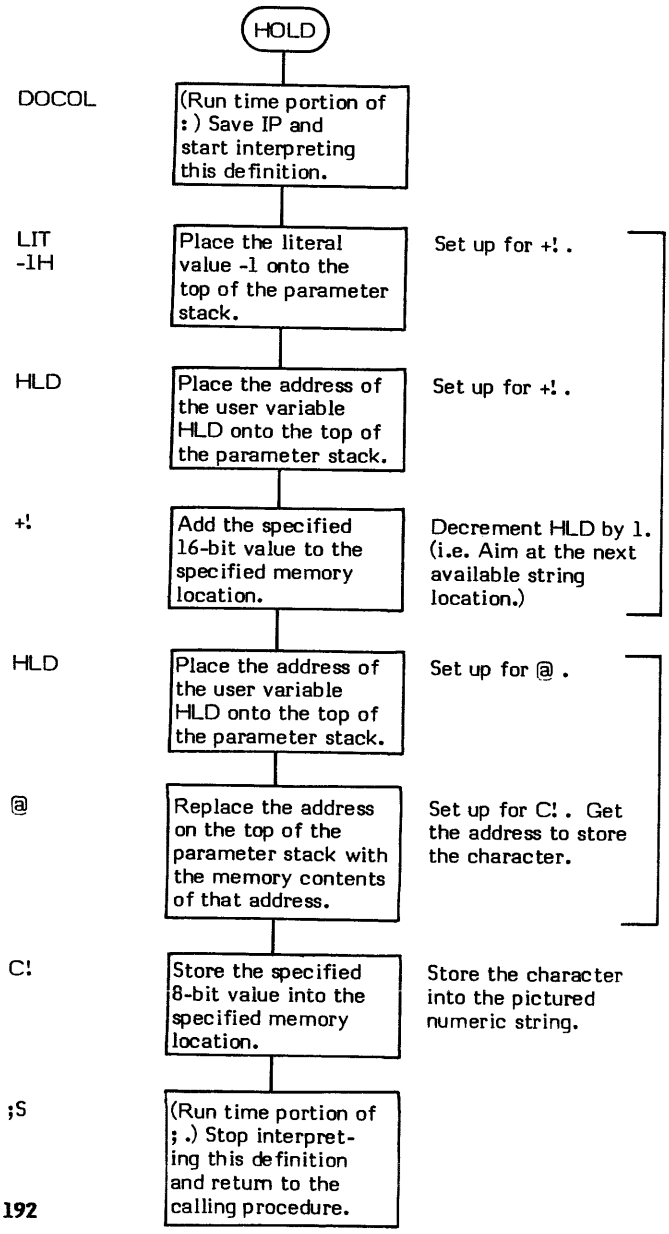
HOLD is a high level colon definition.

Refer to # , <# , #> , SIGN , PAD , and HLD .

FORTH-79: The FORTH-79 equivalent for HOLD is HOLD .

Definition:

```
: HOLD ( char -- )
    -1 HLD +!  HLD @ C! ;
```



HPUSH **(— value)**

HPUSH is strictly an 8080 fig-FORTH Version 1.1 inner interpreter routine entry point.

Entry at this entry point causes the 16-bit contents of the HL register pair to be pushed onto the top of the parameter stack before performing NEXT .

Refer to NEXT .

I

I (— Index value)

I copies the current DO LOOP (or DO +LOOP) Index onto the top of the parameter stack.

This is analogous to the value I in a BASIC FOR-NEXT loop of the form:

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT I
```

In most current FORTH implementations, the loop Index is kept on the top of the return stack. Implementation wise, this makes the code for the word I identical to that of the word R . However, there are no restrictions against keeping the loop Index somewhere else other than the return stack. Therefore, the word I (and not R) should always be used to obtain the current loop Index for transportability. Conversely, the action of I is undefined outside of a DO-LOOP structure.

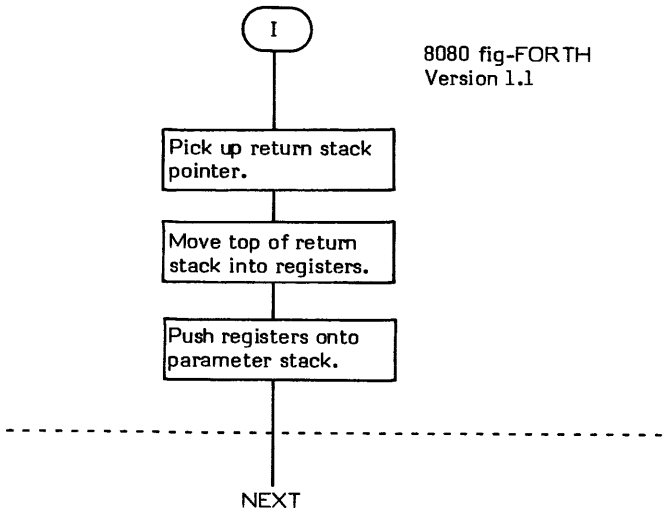
Note that the return stack value is copied; not removed.

LIST is an example of a word which uses I .

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit loop Index value.

I is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for I is I .



ID. **(Name Field Address —)**

ID. (pronounced "I-D-dot") lists the Name Field of a definition.

ID. calculates the physical length of the Name Field by subtracting the Link Field Address from the Name Field Address. This length is then used to move the Name Field text, including the length byte to PAD . The contents of the length byte (which may differ from the actual number of characters stored in the dictionary, see WIDTH) is then used to TYPE the definition name. This may result in the physical Name Field being typed followed by trailing blanks (e.g., A WIDTH of 3 with a length of 7 would result in the first three characters of the name being typed followed by four blanks).

VLIST is an example of a word which uses ID. .

* **At entry** - The top of the parameter stack contains the Name Field Address of the Name Field to be printed.
* **At exit** - No parameters.

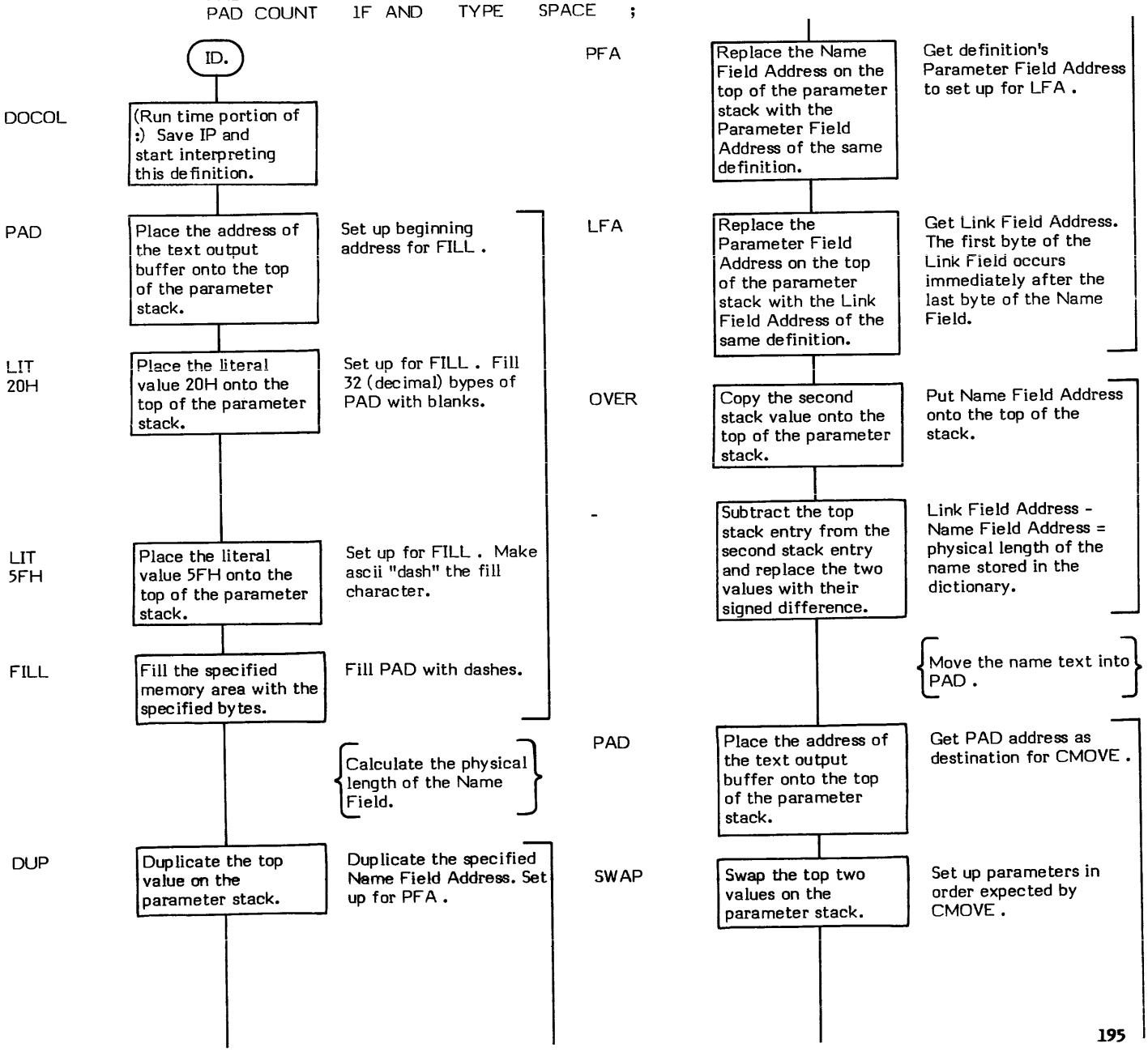
ID. is a high level colon definition.

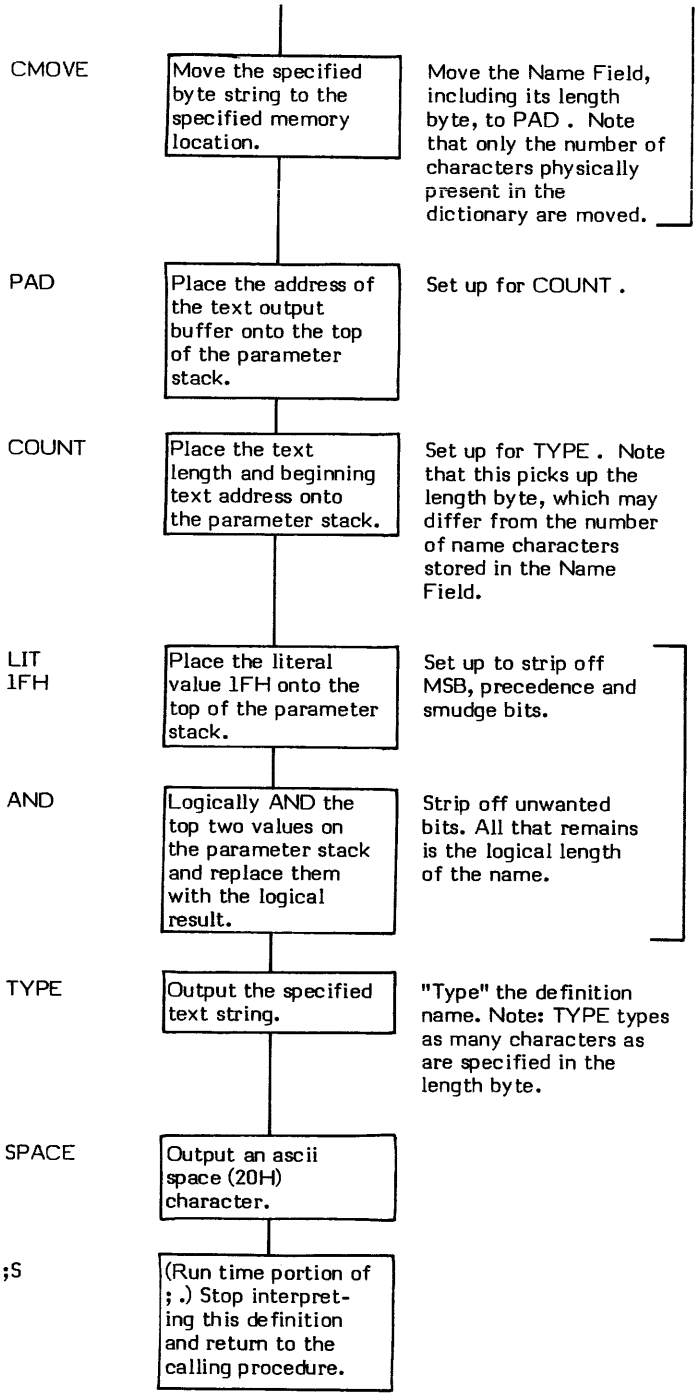
Refer to WIDTH .

FORTH-79: There is no FORTH-79 equivalent for ID. .

Definition:

```
: ID.  ( NFA -- )
    PAD  20 5F FILL  DUP PFA LFA  OVER -  PAD SWAP CMOVE
    PAD COUNT  1F AND  TYPE  SPACE  ;
```




CMOVE
Move the specified byte string to the specified memory location.
Move the Name Field, including its length byte, to PAD . Note that only the number of characters physically present in the dictionary are moved.
PAD
Place the address of the text output buffer onto the top of the parameter stack.
Set up for COUNT .
COUNT
Place the text length and beginning text address onto the parameter stack.
Set up for TYPE . Note that this picks up the length byte, which may differ from the number of name characters stored in the Name Field.
LIT 1FH
Place the literal value 1FH onto the top of the parameter stack.
Set up to strip off MSB, precedence and smudge bits.
AND
Logically AND the top two values on the parameter stack and replace them with the logical result.
Strip off unwanted bits. All that remains is the logical length of the name.
TYPE
Output the specified text string.
"Type" the definition name. Note: TYPE types as many characters as are specified in the length byte.
SPACE
Output an ascii space (20H) character.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

IF

COMPILE TIME (Sequence 2): (— offset address \2)

EXECUTION TIME (Sequence 3): (truth flag —)

IF is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

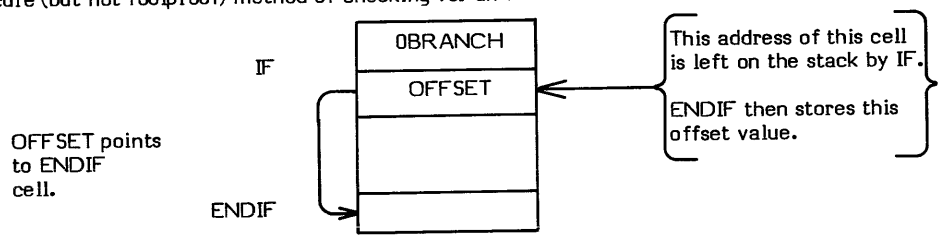
IF must be used within an IF-THEN or IF-ELSE-THEN structure. (ENDIF may be used in place of THEN if desired but THEN is the preferred usage.) The format of these structures is:

IF "true portion" THEN
OR
IF "true portion" ELSE "false portion" THEN

(Beware — This differs in form from the PASCAL IF-THEN-ELSE structure.)

IF-THEN and IF-ELSE-THEN structures must be used within a colon definition.

The compile time (Sequence 2) action of IF is to compile a 0BRANCH into the definition and then reserve the dictionary location following for the branch offset used as an input to 0BRANCH . IF then places the address of this reserved location onto the parameter stack so that a following ELSE or THEN statment can calculate and store its entry point offset into the reserved location. To provide compiler security, the value 2 is placed onto the top of the parameter stack so that ELSE or THEN (or ENDIF) can check for it. This provides a somewhat secure (but not foolproof) method of checking for un-matched IF 's and THEN 's.



The apparent execution time (Sequence 3) action of IF is to input a boolean flag from the top of the parameter stack and take action based on this flag. (Actually 0BRANCH is executed at execution time.) If the flag is true (non-zero), the "true portion" of the structure will be executed. Note that a true flag is any non-zero value; not just a 1. If the flag is false (0), the optional ELSE ("false portion" of the structure) will be executed if it is present. If an ELSE is not included in the structure, a false flag causes control to be passed to the word immediately following the THEN statement.

After executing either the "true portion" or the "false portion" of the structure, control is passed to the word immediately following the THEN statement.

Note that IF is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit single precision value 2. This value is checked by ELSE or THEN to provide compiler security. The second stack entry contains the 16-bit address of the location reserved for the 0BRANCH offset. ELSE or THEN use this address to fill in the offset.

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains a 16-bit signed boolean truth flag.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) — This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) — There is some sort of problem with the pairing of conditionals within the definition being compiled.

IF is a high level colon definition.

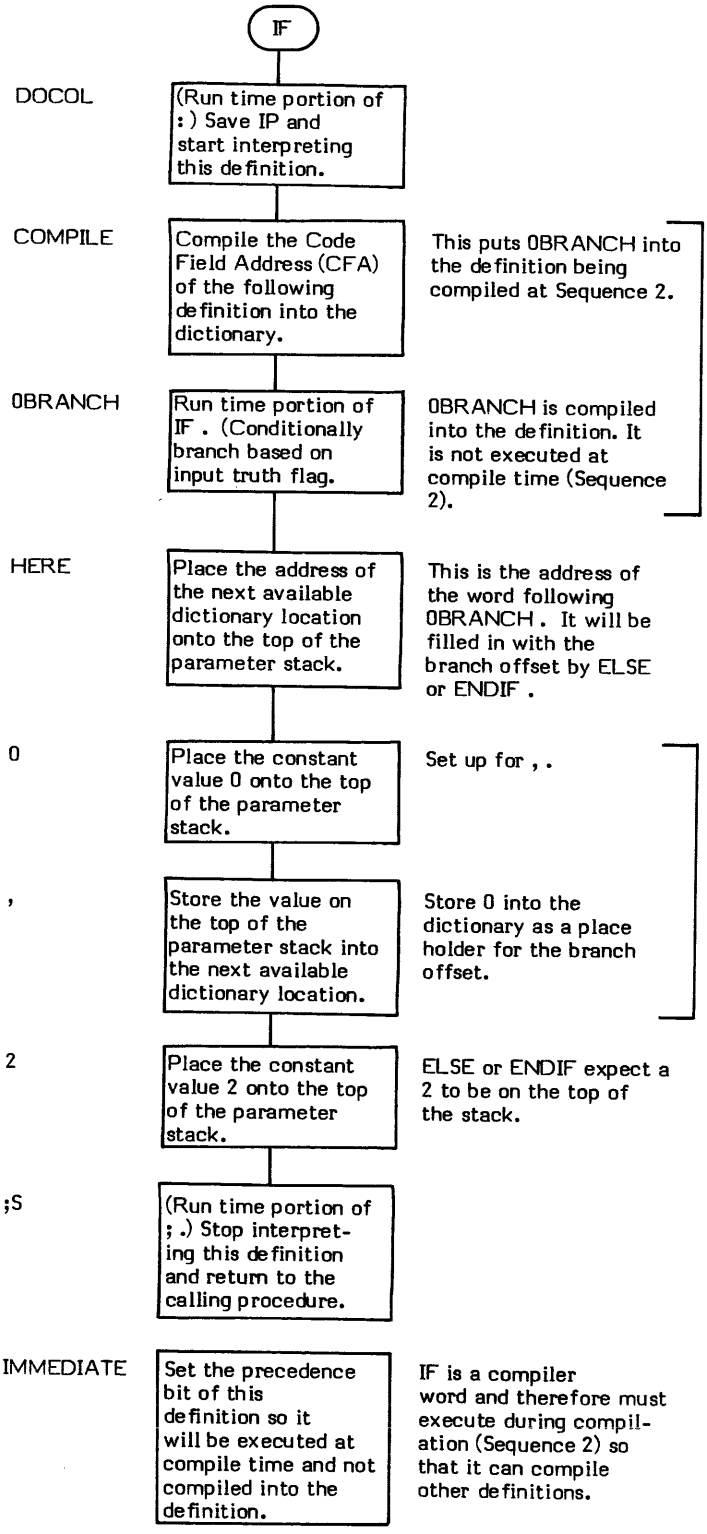
Refer to ELSE , ENDIF , THEN , IF , and 0BRANCH .

FORTH-79: The FORTH-79 equivalent for IF is IF .

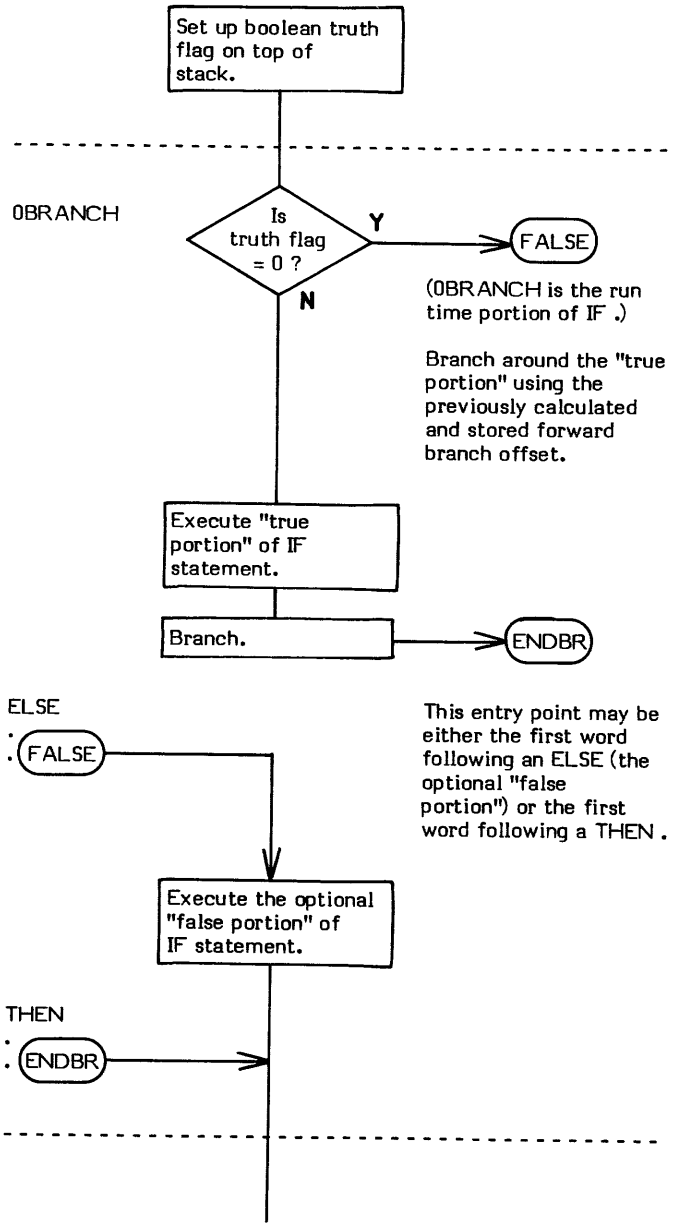
Definition:

```
:   IF   ( — offset address\ 2 )   ( compile time )
        COMPILE 0BRANCH   HERE 0 ,      2   ;   IMMEDIATE
```

COMPILE TIME action of IF (Sequence 2): (— offset address \ 2)



EXECUTION TIME action of IF (Sequence 3): (truth flag —)



IMMEDIATE (--)

IMMEDIATE sets the precedence bit of the most recently created definition. A definition whose precedence bit is set will be executed "immediately" by INTERPRET even when the system is in compile state (hence the name IMMEDIATE).

This feature allows the compiler to be "extended". That is, definitions can be created to perform specific actions at compile time rather than modifying the compiler itself. INTERPRET is the word which detects that the precedence bit is set.

It is possible to force the compilation of an IMMEDIATE definition by preceding it with [COMPILE] . (Otherwise the definition executes instead of compiles!)

The precedence bit in fig-FORTH is the 40H bit of the Name Field's length byte. The system is defined as being in compilation state when the user variable STATE contains a non-zero value.

BEGIN is an example of a compiler word (a conditional compiler in this case) which is flagged as IMMEDIATE by being followed by the word IMMEDIATE .

* **At entry** - No parameters.
* **At exit** - No parameters.

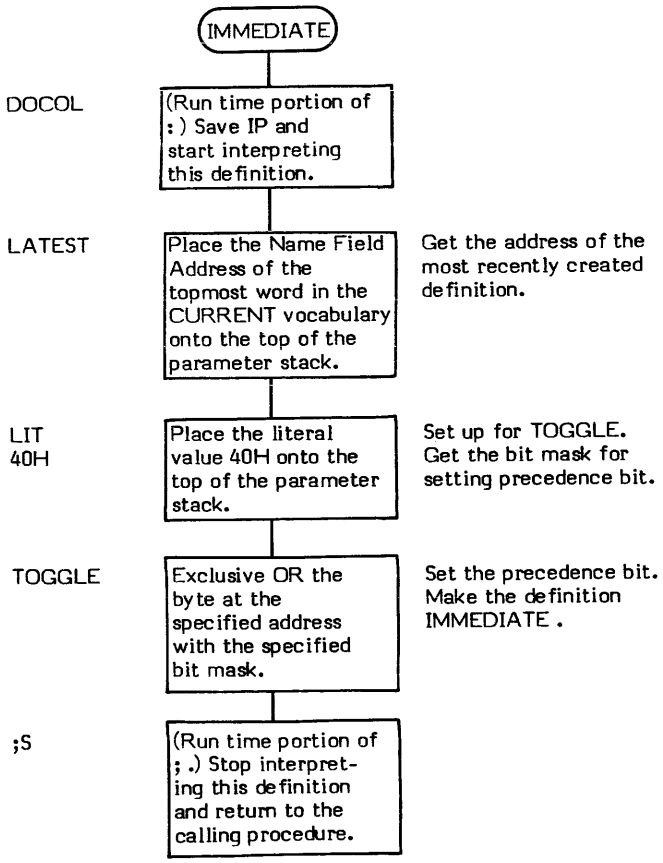
IMMEDIATE is a high level colon definition.

Refer to INTERPRET , [COMPILE] , LASTEST , and STATE .

FORTH-79: The FORTH-79 equivalent for IMMEDIATE is IMMEDIATE .

Definition:

```
:  IMMEDIATE  ( -- )
       LATEST  40  TOGGLE  ;
```



IN

IN (— data address)

IN (pronounced "in") is a user variable that contains the byte offset from the beginning of the current input text buffer (whether using the Terminal Input Buffer or a disk buffer).

IN is set to 0 by QUERY (for TIB input) and by LOAD and --> (for mass storage input). WORD then references and increments IN as words in the buffer are moved to HERE .

The user variable IN is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable IN .

Refer to WORD , QUERY , LOAD , --> , HERE , and USER .

FORTH-79: The FORTH-79 equivalent for IN is >IN (pronounced "to-in") .

INDEX (**beginning screen number \ ending screen number** —)

INDEX prints the first line of each screen within the specified range of screens inclusively. The index listing will be terminated if any terminal key is pressed. See ?TERMINAL .

The basis of INDEX is a DO-LOOP which contains the word .LINE . The loop Index (I) is used as an input parameter to .LINE to print the first line of each screen within the specified range.

* **At entry** - The top of the parameter stack contains an unsigned 16-bit value specifying the last screen to be indexed. The second stack entry contains an unsigned 16-bit value specifying the beginning screen number to be indexed.

* **At exit** - No parameters.

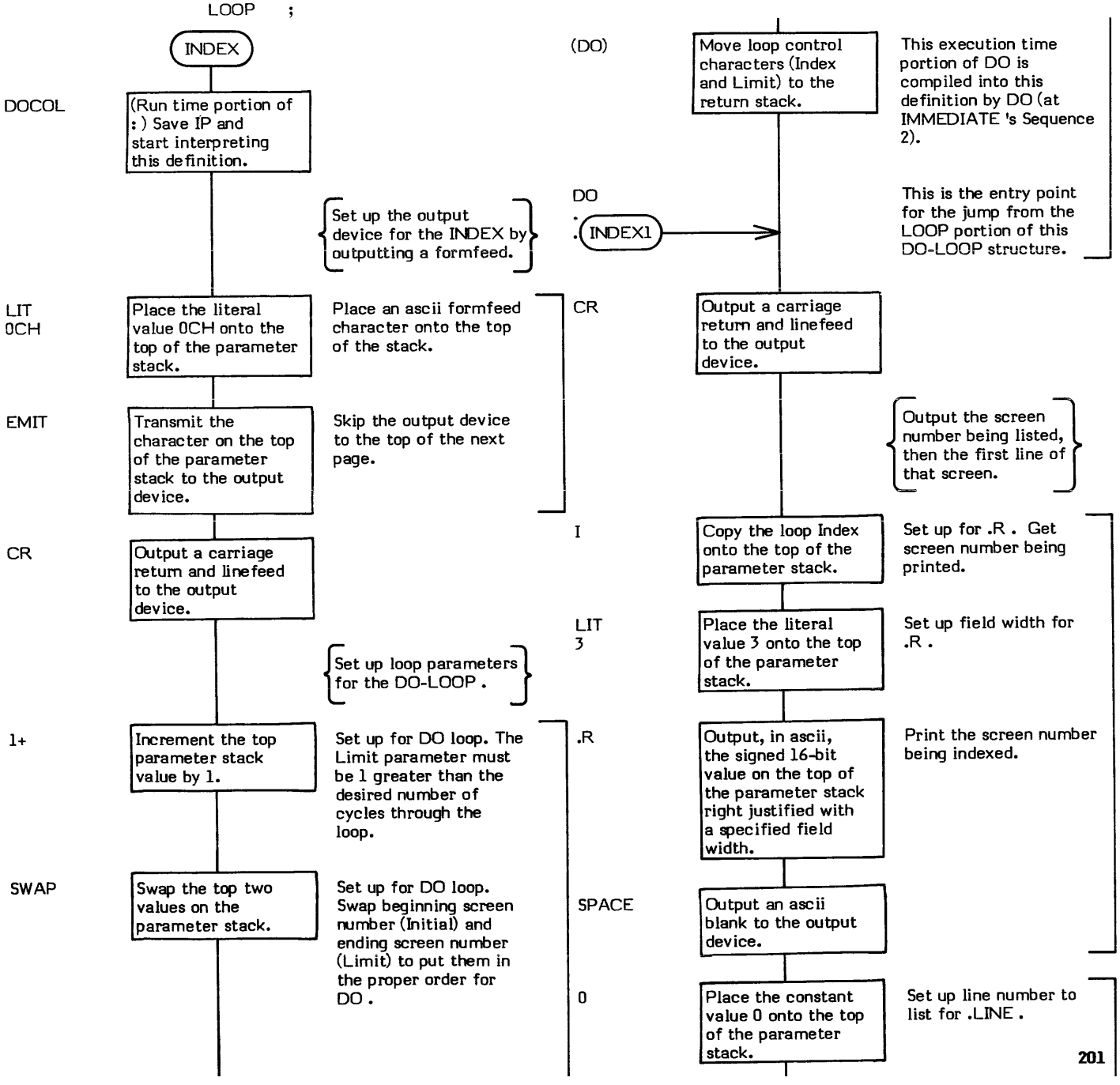
INDEX is a high level colon definition.

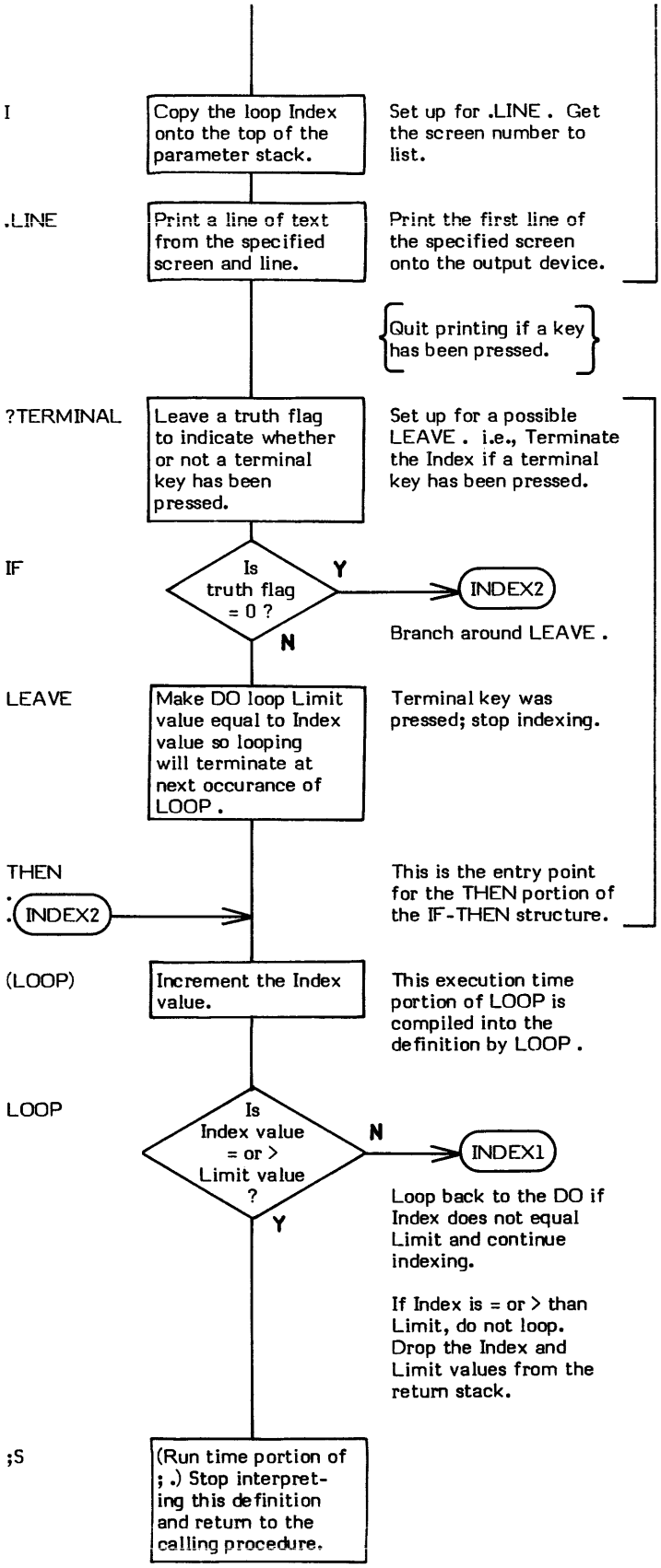
Refer to .LINE , and ?TERMINAL .

FORTH-79: INDEX is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:

```
: INDEX ( beginning screen number\ ending screen number )
    0C EMIT    CR    1+ SWAP
    DO
        CR    I 3 .R SPACE    0 I .LINE
        ?TERMINAL    IF    LEAVE    THEN
    LOOP    ;
```




I
Copy the loop Index onto the top of the parameter stack.
Set up for .LINE . Get the screen number to list.
.LINE
Print a line of text from the specified screen and line.
Print the first line of the specified screen onto the output device.
Quit printing if a key has been pressed.
?TERMINAL
Leave a truth flag to indicate whether or not a terminal key has been pressed.
Set up for a possible LEAVE . i.e., Terminate the Index if a terminal key has been pressed.
IF
Is truth flag = 0 ?
Y
INDEX2
N
Branch around LEAVE .
LEAVE
Make DO loop Limit value equal to Index value so looping will terminate at next occurance of LOOP .
Terminal key was pressed; stop indexing.
THEN
INDEX2
This is the entry point for the THEN portion of the IF-THEN structure.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
INDEX1
Y
Loop back to the DO if Index does not equal Limit and continue indexing.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

INTERPRET

INTERPRET is the FORTH text interpreter, commonly referred to as the outer interpreter. (The address interpreter, NEXT , is referred to as the inner interpreter.)

The actions of INTERPRET are very straightforward. INTERPRET interprets the text input stream. This input stream comes from the terminal if BLK contains 0, or from mass storage if BLK contains a block number.

INTERPRET will either execute (i.e., "interpret" in the traditional sense) or compile an input stream word depending upon the "state" of the system (as reflected by the value stored in the user variable STATE). STATE is set to compile mode by a compiler word such as : and reset to interpret mode by a word such as ; .

The overall logic of INTERPRET is to select the next sequential word in the input stream and search the CONTEXT then CURRENT vocabularies for a match. If no match is found, an attempt is made to convert the "word" to a numeric value (using the current base). If this also fails, an error message is issued.

If -FIND does find a match, the definition is either executed or compiled. Tests are made to determine if the word is IMMEDIATE or if the system is in interpret mode. The word is EXECUTEd if either are true; otherwise, it is compiled into the dictionary.

If a match was not found but the word was successfully converted into a numeric value, DPL is examined to determine if the value is to be treated as double or single precision. By convention, if a decimal point is encountered anywhere within a character string representing a valid numeric value, that value is treated as being double precision. Note that the only function the decimal point serves is as a flag to signify double precision. No "place holding" is implied. (The user variable DPL is used to loosely keep track of the decimal point location. See DPL , and NUMBER .)

Then either DLITERAL or LITERAL is executed. These words test STATE internally and will either leave the value on the stack or compile it depending upon the state of the system.

INTERPRET examines the parameter stack pointer to determine if it is within its maximum and minimum address limits. If not, Error Message 1 (EMPTY STACK) or Error Message 7 (FULL STACK) is issued and a QUIT occurs.

INTERPRET is an endless BEGIN-AGAIN loop and therefore has no way to terminate execution and exit back to the definition which "called" it. This peculiarity is solved via the word NULL (or X).

INTERPRET is used within two definitions: QUIT and LOAD . Data stream input when executed within QUIT comes from the terminal input buffer (TIB). Data stream input from LOAD comes from disk buffers.

It is mandatory (and automatic unless a bug occurs elsewhere) that both of these buffers are ended with an ascii "null" (00) character, hence the definition name NULL . The Name Field of NULL actually contains an ascii null, so when the end of a buffer is encountered, -FIND searches the dictionary for NULL and executes it. NULL drops the top value from the return stack and then returns to the definition following INTERPRET ; not back to INTERPRET .

The reason why the user variable STATE contains a C0H to denote compilation mode is covered in the decription of] .

* **At entry** - No stack parameters but the text input must be formatted such that tokens (words) are separated by blanks. Buffers must be terminated with ascii nulls (00).

* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

? pronounced "HUH"? (0) -- The word in question cannot be found in the dictionary and is not a number.

EMPTY STACK (1) -- More values have been removed from the parameter stack than were added.

FULL STACK (7) -- Too many values have been added to the parameter stack.

INTERPRET is a high level colon definition.

Refer to WORD , -FIND , NULL (or X), DPL , QUIT , LOAD , and] .

FORTH-79: INTERPRET is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:

```
: INTERPRET ( – )
    BEGIN
        -FIND IF
                STATE @ < IF
                            CFA ,
                          ELSE
                            CFA EXECUTE
                          THEN
                ?STACK
              ELSE
                HERE NUMBER DPL @ 1+ IF
                                       [COMPILE] DLITERAL
                                     ELSE
                                       DROP [COMPILE] LITERAL
                                     THEN
                ?STACK
              THEN
    AGAIN ;
```

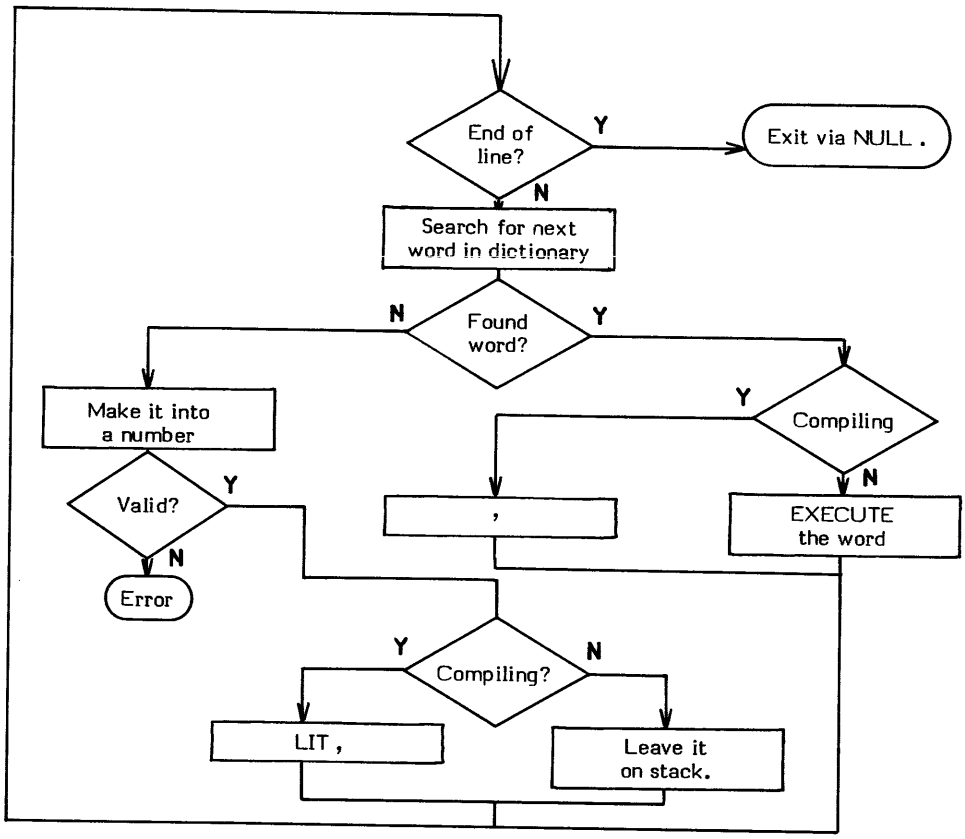
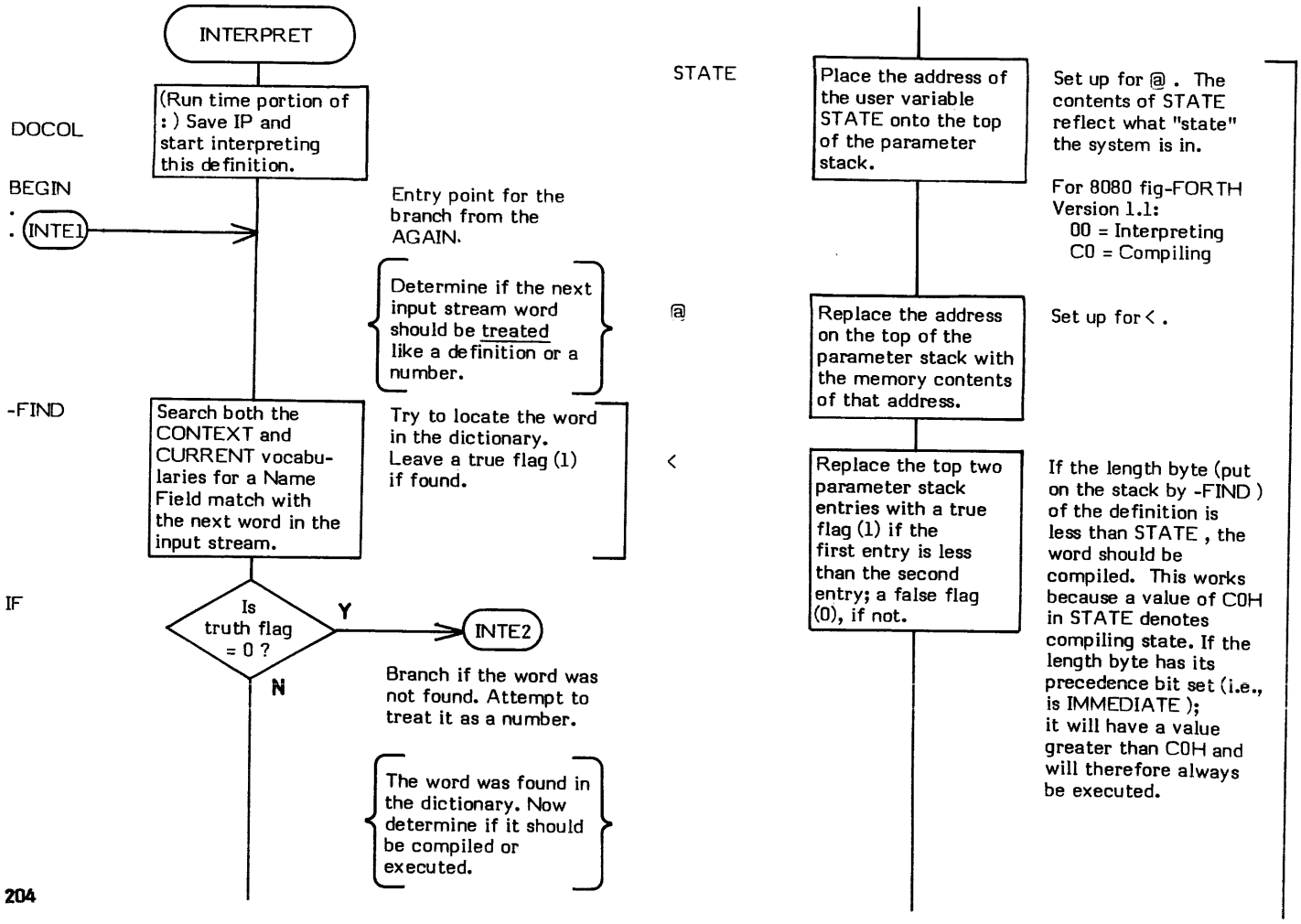
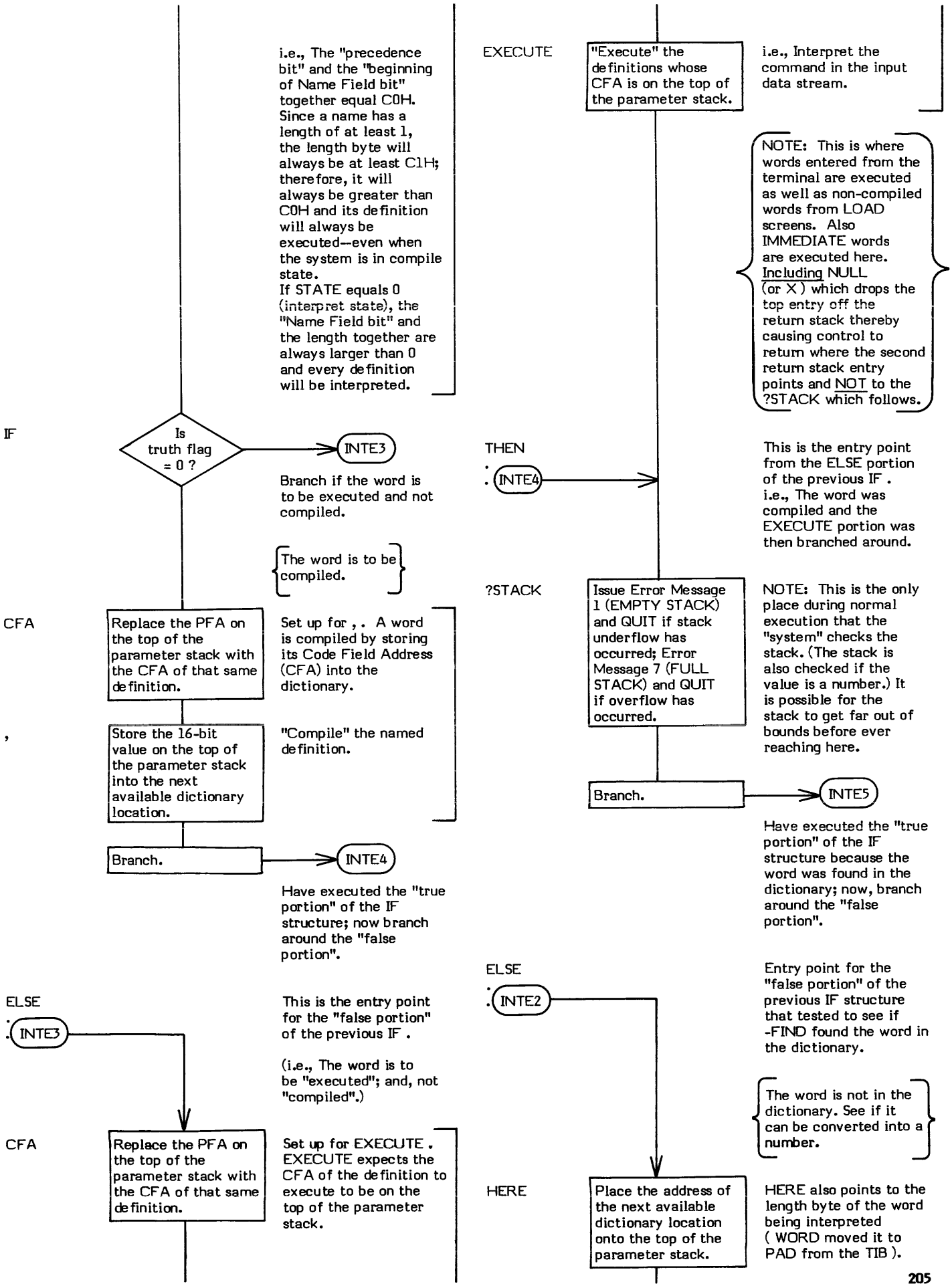
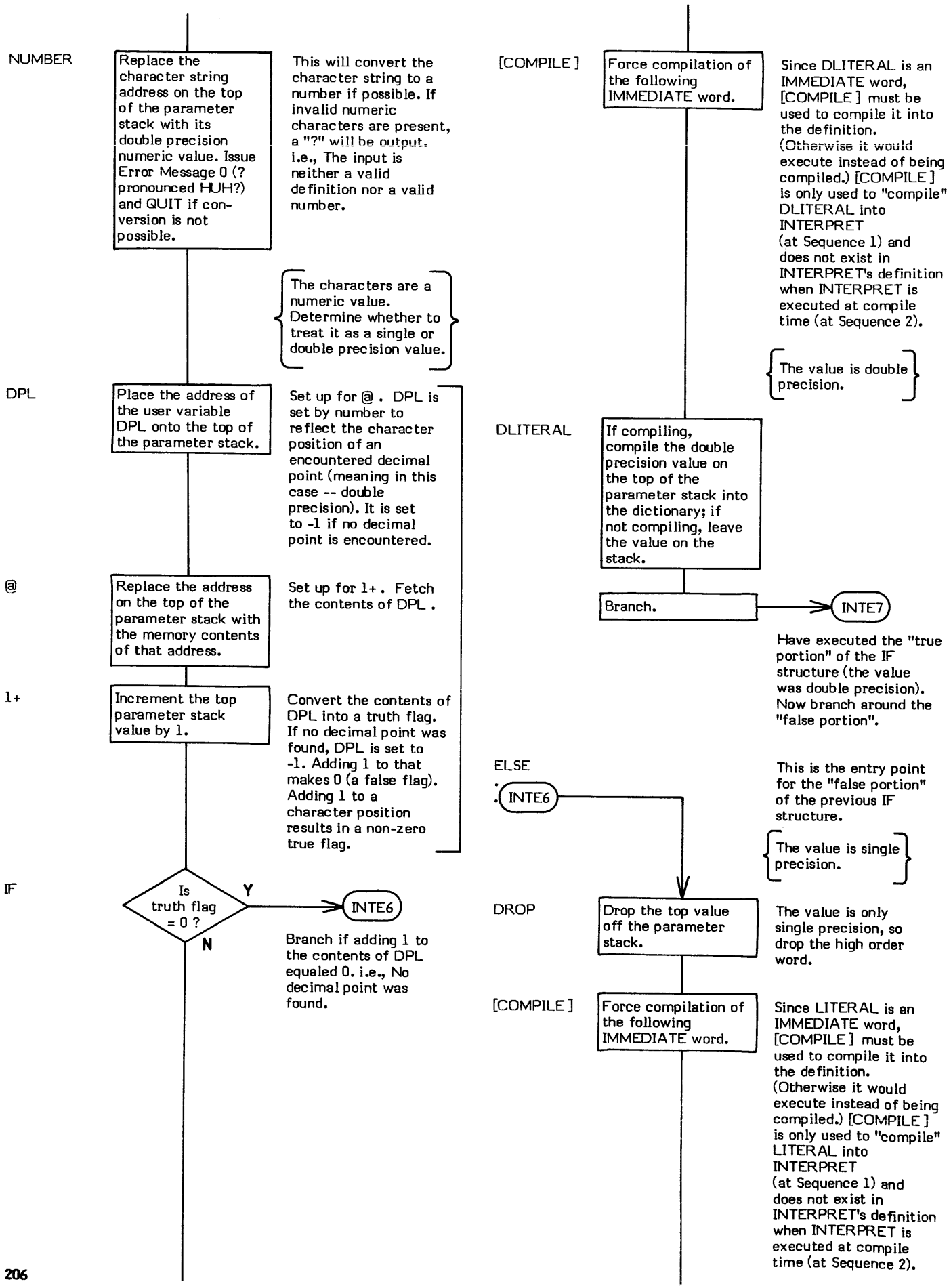


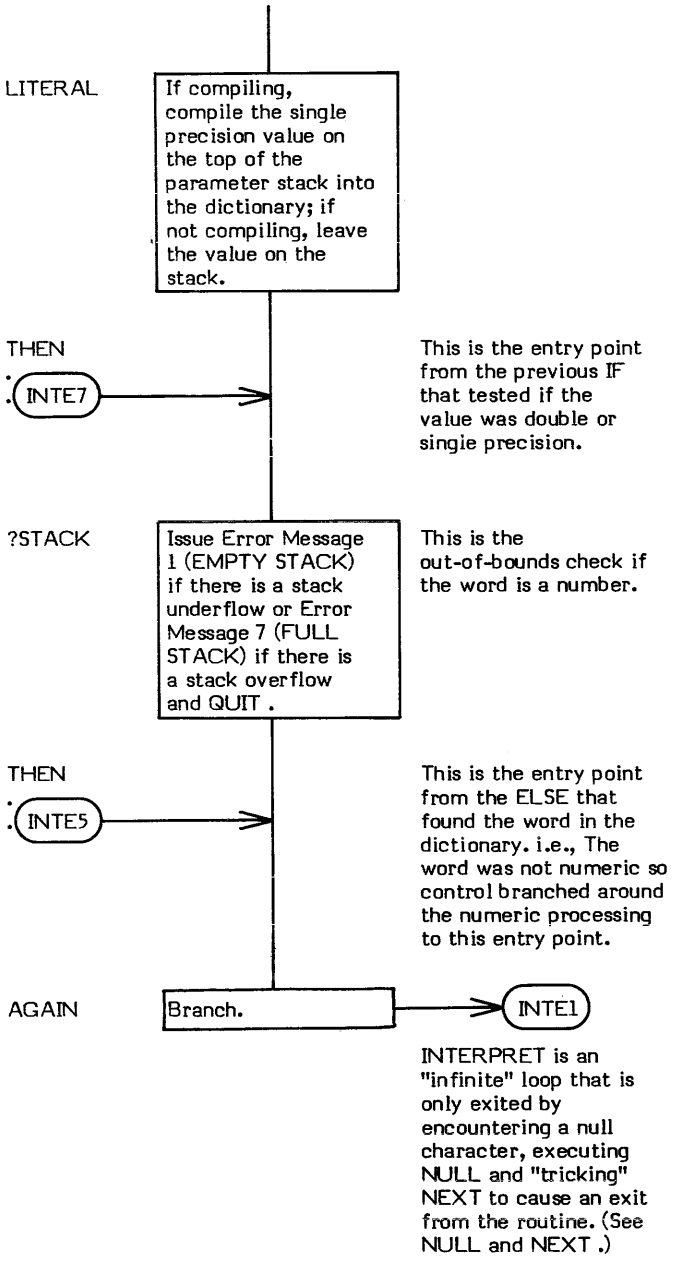
Figure INTERPRET-1

High Level Flow Chart of INTERPRET .




i.e., The "precedence bit" and the "beginning of Name Field bit" together equal C0H. Since a name has a length of at least 1, the length byte will always be at least C1H; therefore, it will always be greater than C0H and its definition will always be executed--even when the system is in compile state.
If STATE equals 0 (interpret state), the "Name Field bit" and the length together are always larger than 0 and every definition will be interpreted.
IF
Is truth flag = 0 ?
INTE3
Branch if the word is to be executed and not compiled.
The word is to be compiled.
CFA
Replace the PFA on the top of the parameter stack with the CFA of that same definition.
Set up for , . A word is compiled by storing its Code Field Address (CFA) into the dictionary.
,
Store the 16-bit value on the top of the parameter stack into the next available dictionary location.
"Compile" the named definition.
Branch.
INTE4
Have executed the "true portion" of the IF structure; now branch around the "false portion".
ELSE
INTE3
This is the entry point for the "false portion" of the previous IF .
(i.e., The word is to be "executed"; and, not "compiled".)
CFA
Replace the PFA on the top of the parameter stack with the CFA of that same definition.
Set up for EXECUTE . EXECUTE expects the CFA of the definition to execute to be on the top of the parameter stack.
EXECUTE
"Execute" the definitions whose CFA is on the top of the parameter stack.
i.e., Interpret the command in the input data stream.
NOTE: This is where words entered from the terminal are executed as well as non-compiled words from LOAD screens. Also IMMEDIATE words are executed here. Including NULL (or X) which drops the top entry off the return stack thereby causing control to return where the second return stack entry points and NOT to the ?STACK which follows.
THEN
INTE4
This is the entry point from the ELSE portion of the previous IF .
i.e., The word was compiled and the EXECUTE portion was then branched around.
?STACK
Issue Error Message 1 (EMPTY STACK) and QUIT if stack underflow has occurred; Error Message 7 (FULL STACK) and QUIT if overflow has occurred.
NOTE: This is the only place during normal execution that the "system" checks the stack. (The stack is also checked if the value is a number.) It is possible for the stack to get far out of bounds before ever reaching here.
Branch.
INTE5
Have executed the "true portion" of the IF structure because the word was found in the dictionary; now, branch around the "false portion".
ELSE
INTE2
Entry point for the "false portion" of the previous IF structure that tested to see if -FIND found the word in the dictionary.
The word is not in the dictionary. See if it can be converted into a number.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
HERE also points to the length byte of the word being interpreted (WORD moved it to PAD from the TIB).


NUMBER
Replace the character string address on the top of the parameter stack with its double precision numeric value. Issue Error Message 0 (? pronounced HUH?) and QUIT if conversion is not possible.
This will convert the character string to a number if possible. If invalid numeric characters are present, a "?" will be output. i.e., The input is neither a valid definition nor a valid number.
The characters are a numeric value. Determine whether to treat it as a single or double precision value.
DPL
Place the address of the user variable DPL onto the top of the parameter stack.
Set up for @ . DPL is set by number to reflect the character position of an encountered decimal point (meaning in this case -- double precision). It is set to -1 if no decimal point is encountered.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for 1+ . Fetch the contents of DPL .
1+
Increment the top parameter stack value by 1.
Convert the contents of DPL into a truth flag. If no decimal point was found, DPL is set to -1. Adding 1 to that makes 0 (a false flag). Adding 1 to a character position results in a non-zero true flag.
IF
Is truth flag = 0 ?
Y
N
INTE6
Branch if adding 1 to the contents of DPL equaled 0. i.e., No decimal point was found.
[COMPILE]
Force compilation of the following IMMEDIATE word.
Since DLITERAL is an IMMEDIATE word, [COMPILE] must be used to compile it into the definition. (Otherwise it would execute instead of being compiled.) [COMPILE] is only used to "compile" DLITERAL into INTERPRET (at Sequence 1) and does not exist in INTERPRET's definition when INTERPRET is executed at compile time (at Sequence 2).
The value is double precision.
DLITERAL
If compiling, compile the double precision value on the top of the parameter stack into the dictionary; if not compiling, leave the value on the stack.
Branch.
INTE7
Have executed the "true portion" of the IF structure (the value was double precision). Now branch around the "false portion".
ELSE
INTE6
This is the entry point for the "false portion" of the previous IF structure.
The value is single precision.
DROP
Drop the top value off the parameter stack.
The value is only single precision, so drop the high order word.
[COMPILE]
Force compilation of the following IMMEDIATE word.
Since LITERAL is an IMMEDIATE word, [COMPILE] must be used to compile it into the definition. (Otherwise it would execute instead of being compiled.) [COMPILE] is only used to "compile" LITERAL into INTERPRET (at Sequence 1) and does not exist in INTERPRET's definition when INTERPRET is executed at compile time (at Sequence 2).


LITERAL
If compiling, compile the single precision value on the top of the parameter stack into the dictionary; if not compiling, leave the value on the stack.
THEN
INTE7
This is the entry point from the previous IF that tested if the value was double or single precision.
?STACK
Issue Error Message 1 (EMPTY STACK) if there is a stack underflow or Error Message 7 (FULL STACK) if there is a stack overflow and QUIT .
This is the out-of-bounds check if the word is a number.
THEN
INTE5
This is the entry point from the ELSE that found the word in the dictionary. i.e., The word was not numeric so control branched around the numeric processing to this entry point.
AGAIN
Branch.
INTE1
INTERPRET is an "infinite" loop that is only exited by encountering a null character, executing NULL and "tricking" NEXT to cause an exit from the routine. (See NULL and NEXT .)

IP

IP

IP is a pointer used by FORTH's "threading" words. It plays an important role in such words as NEXT , ;S , DOCOL (the execution time portion of :), etc.

IP generally serves as a pointer to the next "word" (actually a CFA within the Parameter Field of a definition) to execute. NEXT jumps indirectly "through" this pointer to execute the Code Field procedure.

IP is not a true FORTH word. It is not a variable. It is a logical entity and may be physically kept in registers or memory or whatever depending upon the exact system implementation.

In the 8080 fig-FORTH Version 1.1, IP is contained in the register pair BC.

Refer to NEXT , : , ;S , DOES> , and EXECUTE .

KEY (— input value)

KEY inputs a character from the terminal and places it onto the top of the parameter stack.

NOTE: Control is not returned from KEY until an input character is available. If processing must be performed while waiting for an input character, ?TERMINAL may be used to determine if input data is actually present before invoking KEY.

The actual operation of the word is installation dependent.

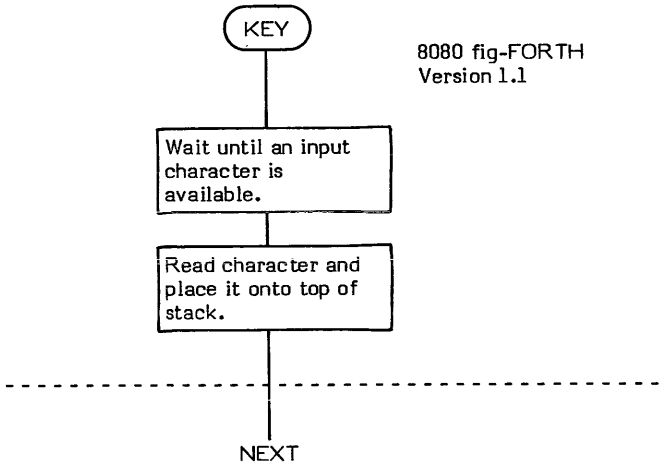
EXPECT is an example of a word which uses KEY .

* **At entry** - No parameters.
* **At exit** -The top of the parameter stack contains a 16-bit value of which the low order portion is the 8-bit value input from the terminal.

KEY is a low level code primitive.

Refer to ?TERMINAL .

FORTH-79: The FORTH-79 equivalent for KEY is KEY .



LATEST

LATEST (-- address)

LATEST places the Name Field Address of the bottom-most word in the CURRENT vocabulary onto the top of the parameter stack. When compiling, this bottom-most word is also the "latest" or most recently compiled definition.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the Name Field Address of the topmost word in the CURRENT vocabulary.

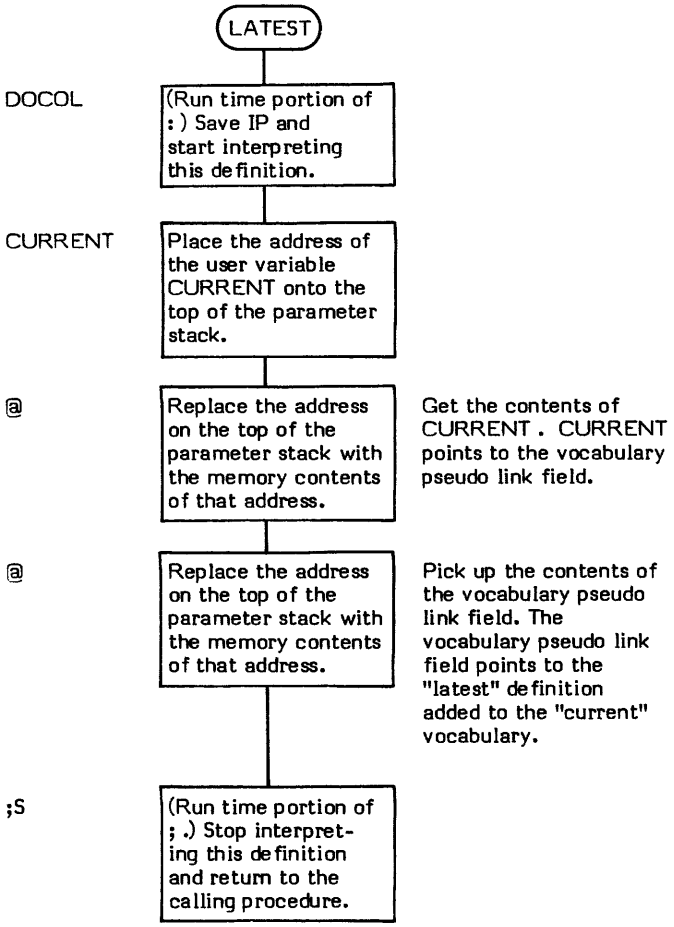
LATEST is a high level colon definition.

Refer to CURRENT , and VOCABULARY .

FORTH-79: There is no FORTH-79 equivalent for LATEST .

Definition:

```
: LATEST ( -- address )
      CURRENT @ @ ;
```



LEAVE (–)

LEAVE is used to prematurely exit a DO-LOOP. It forces termination of the loop by setting the Limit value equal to the current value of the Index which in turn causes an exit at the next execution of LOOP or +LOOP . The Index value remains valid and unchanged.

Note that the fact that the Index and Limit values are kept on the return stack is an installation dependent choice. (Refer to I).

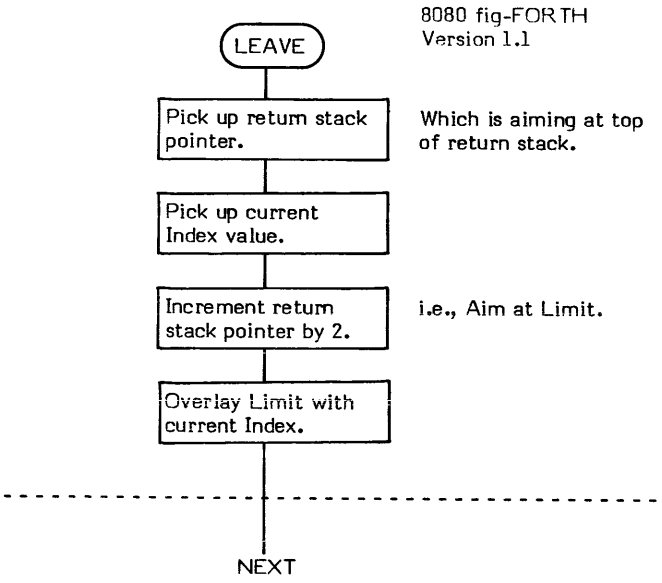
INDEX is an example of a word which uses LEAVE to prematurely exit a DO-LOOP structure.

* **At entry** - No parameter stack entries.
* **At exit** - No parameter stack entries.

LEAVE is a low level code primitive.

Refer to DO , LOOP , +LOOP , (LOOP) , (+LOOP) , and I .

FORTH-79: The FORTH-79 equivalent for LEAVE is LEAVE .



LFA

LFA (Parameter Field Address -- Link Field Address)

LFA (pronounced "L-F-A") converts a given Parameter Field Address of a dictionary definition into its Link Field Address.

The structure of the header of a FORTH definition is:

Name Field	Variable length
Link Field	2 byte address pointer
Code Field	2 byte address pointer
Parameter Field	Variable length

An example of the use of LFA can be found in the word ID. .

* **At entry** - The top of the parameter stack contains the Parameter Field Address of a FORTH definition.
* **At exit** - The top of the parameter stack contains the 16-bit Link Field Address of the specified FORTH definition.

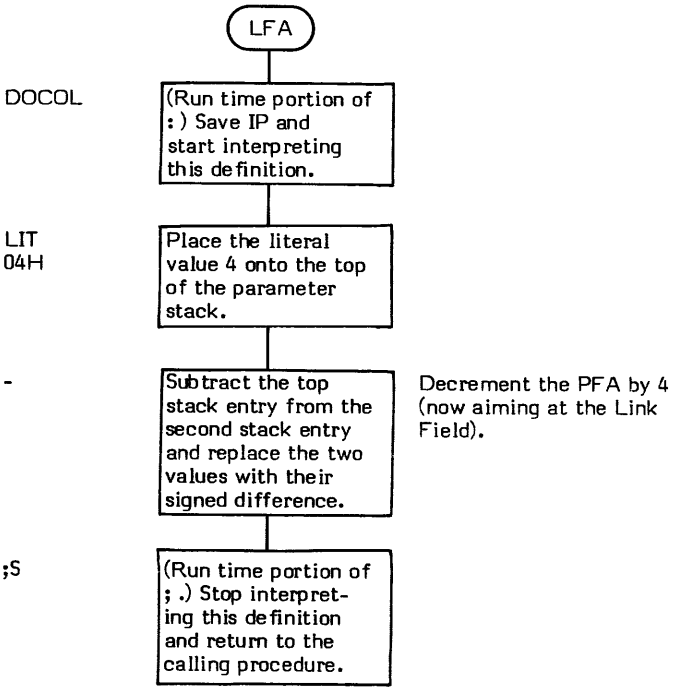
LFA is a high level colon definition.

Refer to NFA , CFA , and PFA .

FORTH-79: There is no FORTH-79 equivalent for LFA .

Definition:

```
: LFA ( PFA -- LFA )
    04 - ;
```



LIMIT (— address)

LIMIT is a single precision CONSTANT value. This constant places the memory address of the next memory location past the end of the last or highest (hence " LIMIT ") block buffer in the buffer-array onto the top of the parameter stack. Refer to +BUF for an example of the usage of LIMIT . Also see BLOCK and BUFFER for descriptions of the buffer-array.

Conversely, FIRST is the constant that reflects the lowest (hence " FIRST ") buffer-array address.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the signed 16-bit memory address of the next memory location after the end of the last block buffer in the buffer-array.

Refer to +BUF , BLOCK , BUFFER , and FIRST .

FORTH-79: There is no FORTH-79 equivalent for LIMIT .

LIST

LIST (Screen number —)

LIST "lists" a specified screen onto the output device. The value of the constant C/L (Characters per Line) determines the length of an editing screen line. A screen is usually divided into 16 lines of 64 characters each.

The basis of LIST is a DO-LOOP which uses the loop Index as an input to .LINE .

TRIAD is an example of a word which uses LIST .

* **At entry** - The top of the parameter stack contains the unsigned 16-bit screen number to be listed.
* **At exit** - No parameters on the stack. However, the user variable SCR now contains the screen number that was listed.

LIST is a high level colon definition.

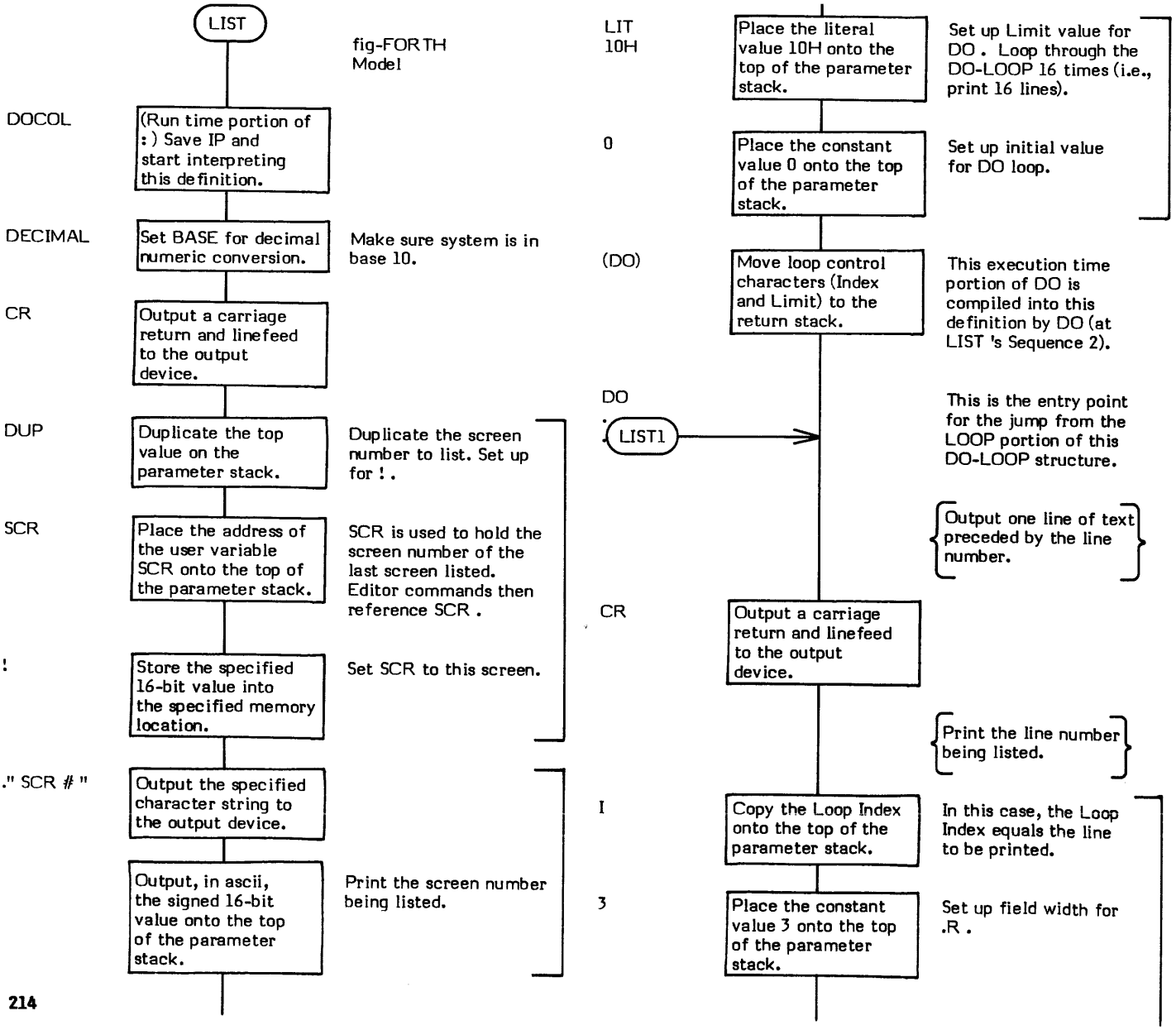
Refer to C/L , and B/SCR .

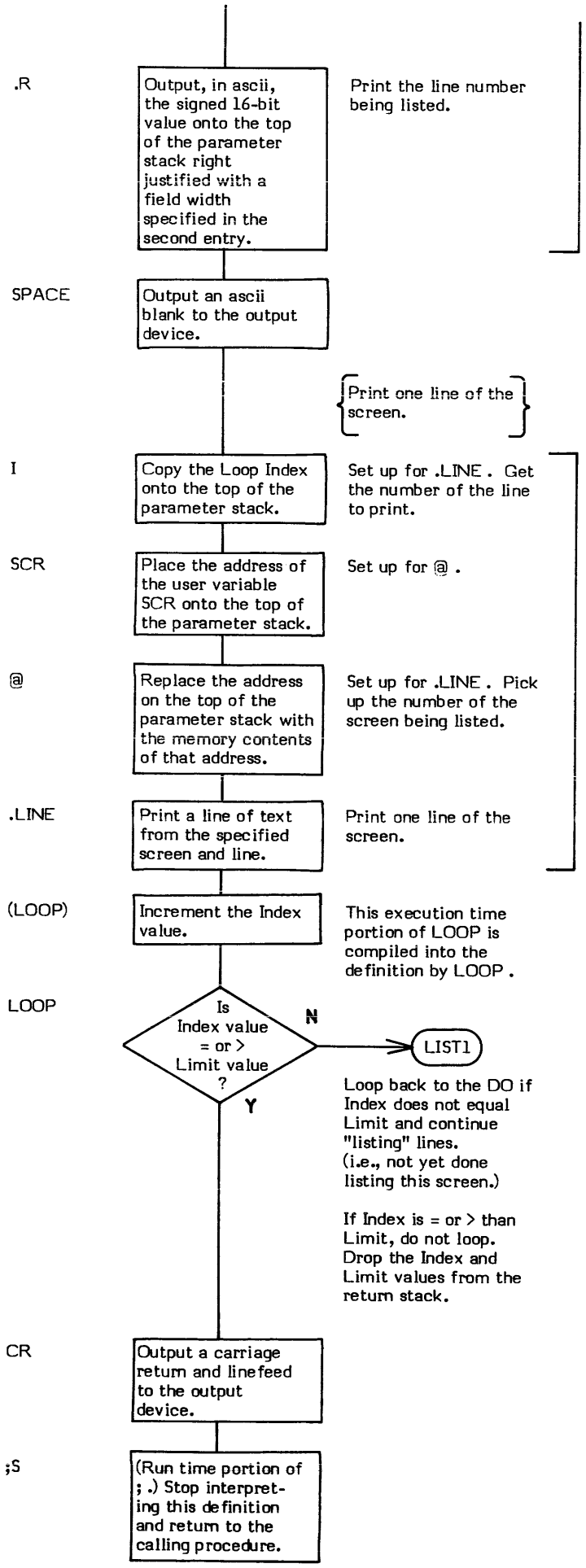
FORTH-79: The FORTH-79 equivalent for LIST is LIST .

NOTE: The 8080 fig-FORTH Version 1.1 has been changed to stop listing if a key is pressed. The following flowchart shows the main logic of LIST according to the fig-FORTH Model.

Definition:

```
:  LIST  ( screen number -- )
      DECIMAL  CR  DUP SCR !  ." SCR" .  10 0
      DO
         CR I 3 .R  SPACE  I SCR @ .LINE
      LOOP  CR  ;
```




.R
Output, in ascii, the signed 16-bit value onto the top of the parameter stack right justified with a field width specified in the second entry.
Print the line number being listed.
SPACE
Output an ascii blank to the output device.
Print one line of the screen.
I
Copy the Loop Index onto the top of the parameter stack.
Set up for .LINE . Get the number of the line to print.
SCR
Place the address of the user variable SCR onto the top of the parameter stack.
Set up for @ .
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for .LINE . Pick up the number of the screen being listed.
.LINE
Print a line of text from the specified screen and line.
Print one line of the screen.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
LIST1
Y
Loop back to the DO if Index does not equal Limit and continue "listing" lines. (i.e., not yet done listing this screen.)
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
CR
Output a carriage return and linefeed to the output device.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

LIT

LIT (— literal value)

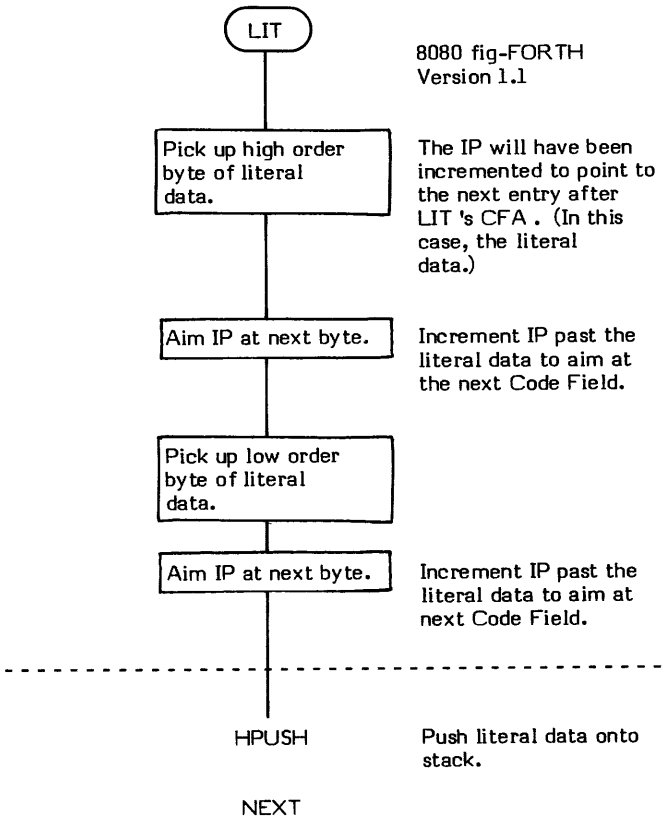
LIT places the 16-bit contents of the next dictionary location onto the top of the paramter stack. LIT , followed by the literal value, is compiled into a definition by LITERAL and DLITERAL . LIT , then, is the execution time (Sequence 3) action of LITERAL and DLITERAL .

* **At entry** - No stack parameters although the desired literal value must reside in the next dictionary entry.
* **At exit** - The top of the parameter stack contains a 16-bit value equal to that contained in the location following LIT .

LIT is a low level code primitive.

Refer to LITERAL , and DLITERAL .

FORTH-79: There is no FORTH-79 equivalent for LIT .



LITERAL

COMPILE TIME (Sequence 2): (value —)

EXECUTION TIME (Sequence 3): (— value)

LITERAL is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

Although LITERAL has no effect (Sequence 2) if not used within a colon definition, it does not signal an error. The compile time action of LITERAL is to compile a dynamically calculated 16-bit value into a definition. The end result of the compile action here is the same as LIT . LIT is compiled into a definition "automatically" (in the Interpreter) by the presence of a 16-bit numeric value in the input stream. However, if this numeric value is calculated dynamically during compilation, it will not exist in the input stream and therefore some other method must be used to compile LIT and the value into the dictionary. LITERAL is this other method.

An example of the use of LITERAL would be calculating the length of a table to be ERASEd. This calculation will be performed only once, at compile time instead of each time the table is erased. In this example, TABLE-LEN and #-OF-ENTRIES , when multiplied together, give the number of bytes to erase. This sample definition would look like this:

```
: ERASE-TABLE  BEG-TABLE-ADDR  [ TABLE-LEN  #-OF-ENTRIES  *  ] LITERAL  ERASE  ;
```

[stops compilation, TABLE-LEN and #-OF-ENTRIES are multiplied, and their product is left on the stack.] resumes compilation and LITERAL then compiles the value into the definition.

The execution time action of LITERAL is that of LIT . That is, upon execution of LIT, the 16-bit value is placed on the top of the parameter stack.

Note that LIT is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

- **At entry** - The top of the parameter stack contains a signed 16-bit single precision value to be compiled into the definition.
- **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

- **At entry** - No parameters.
- **At exit** - The top of the parameter stack contains the signed 16-bit single precision value previously compiled into the definition being executed.

LITERAL is a high level colon definition.

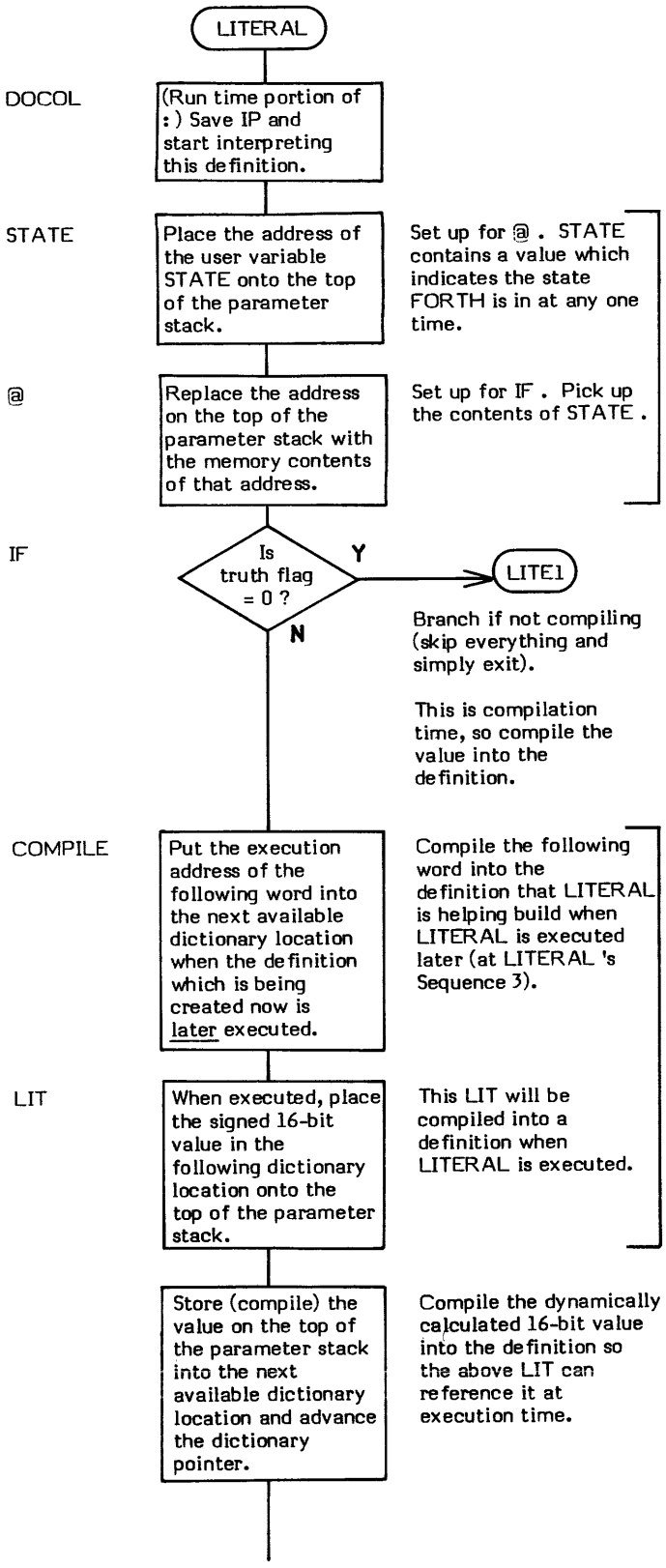
Refer to LIT , [,] , and INTERPRET .

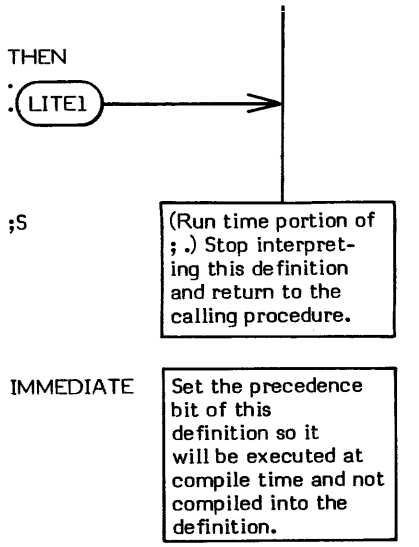
FORTH-79: The FORTH-79 equivalent for LIT is LIT .

Definition:

```
:  LITERAL  ( value -- )  ( compile time )
     STATE @  IF  COMPILE LIT  ,  THEN  ;  IMMEDIATE
```

COMPILE TIME action of LITERAL (Sequence 2): (value —)





EXECUTION TIME action of LITERAL (Sequence 3): (— value)

The EXECUTION time action of LITERAL is that of LIT .

LOAD (screen number --)

LOAD begins interpretation of the source text from the screen number specified on the top of the parameter stack. The blocks that comprise the specified screen are read from mass storage to memory (if they are not already in the buffers) and interpreted.

LOAD starts an entirely new interpretation level, therefore LOADs may be nested. Loading terminates upon encountering a null (0) or at ;S . A null or ;S will pop back up to the previous level, and resume interpretation after the word that caused LOAD to be executed.

Note: "0 LOAD" will cause a return stack overflow and a system crash!

As a matter of programming style, it is often desirable to create a "load screen" which specifically loads each screen via LOAD (with each LOAD followed by a comment describing the screen to be loaded) rather than loading multiple screens via --> . This makes it much easier to keep track of what is being loaded.

* **At entry** - The top of the parameter stack contains a 16-bit unsigned number specifying the screen to "load" and interpret.
* **At exit** - No parameters.

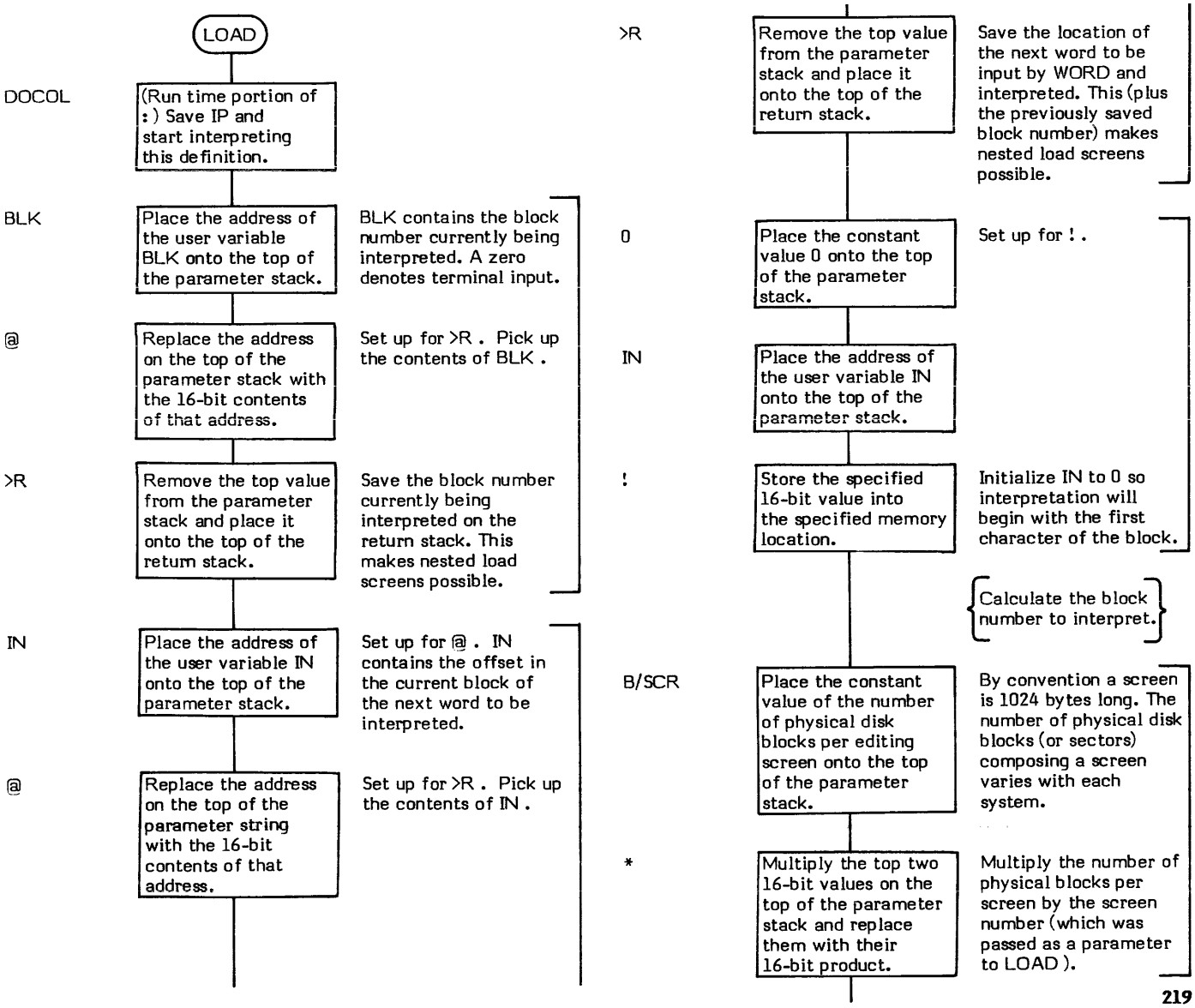
LOAD is a high level colon definition.

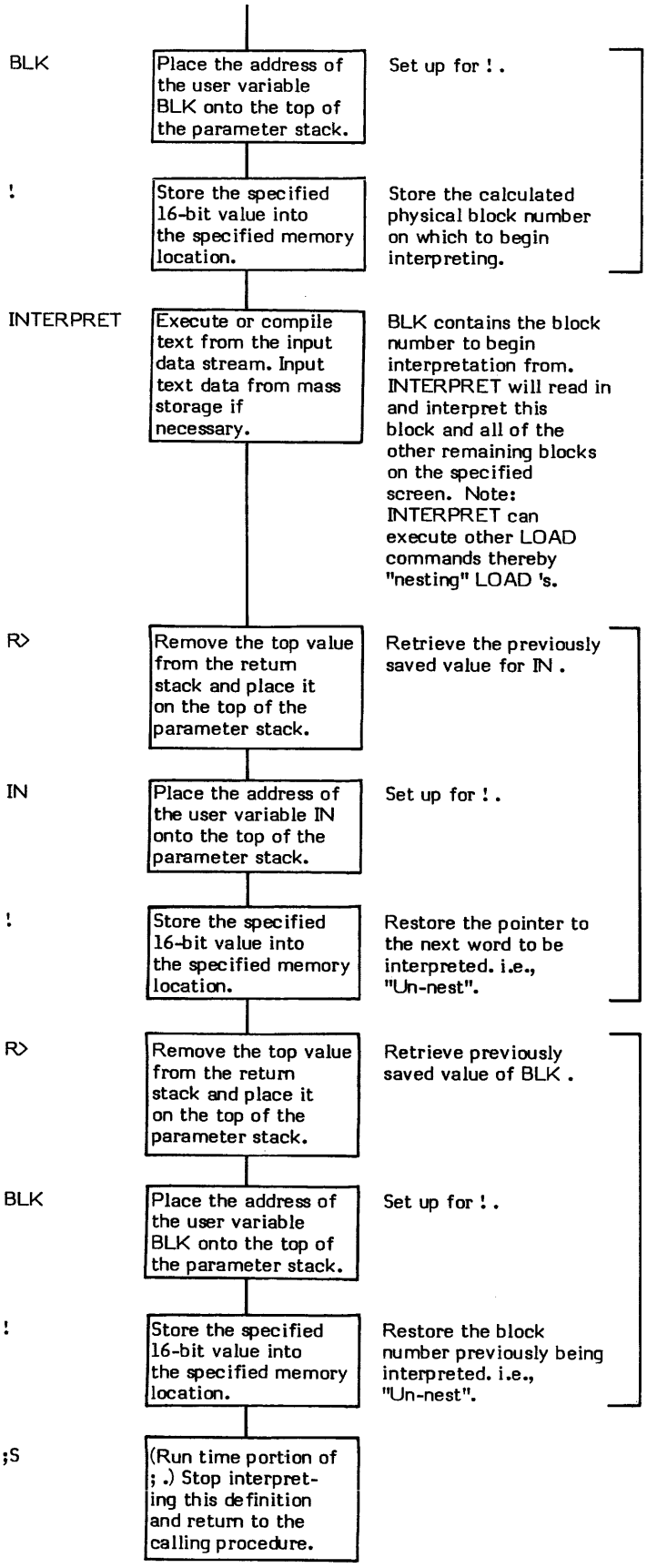
Refer to --> , WORD , and INTERPRET .

FORTH-79: The FORTH-79 equivalent for LOAD is LOAD .

Definition:

```
: LOAD ( screen number -- )
    BLK @ >R   IN @ >R   0 IN !   B/SCR *   BLK !
    INTERPRET   R> IN !   R> BLK !   ;
```




BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for ! .
!
Store the specified 16-bit value into the specified memory location.
Store the calculated physical block number on which to begin interpreting.
INTERPRET
Execute or compile text from the input data stream. Input text data from mass storage if necessary.
BLK contains the block number to begin interpretation from. INTERPRET will read in and interpret this block and all of the other remaining blocks on the specified screen. Note: INTERPRET can execute other LOAD commands thereby "nesting" LOAD 's.
R>
Remove the top value from the return stack and place it on the top of the parameter stack.
Retrieve the previously saved value for IN .
IN
Place the address of the user variable IN onto the top of the parameter stack.
Set up for ! .
!
Store the specified 16-bit value into the specified memory location.
Restore the pointer to the next word to be interpreted. i.e., "Un-nest".
R>
Remove the top value from the return stack and place it on the top of the parameter stack.
Retrieve previously saved value of BLK .
BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for ! .
!
Store the specified 16-bit value into the specified memory location.
Restore the block number previously being interpreted. i.e., "Un-nest".
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

LOOP

COMPILE TIME (Sequence 2): (loop address \3 --)

EXECUTION TIME (Sequence 3): (--)

LOOP is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

LOOP is used to end a DO-LOOP structure in conjunction with the word DO . The compile time action of LOOP is to compile the run time word (LOOP) into the dictionary. Secondly, it resolves the "loop body" entry point address provided by DO into a return branch offset used by (LOOP) and stores this offset into the dictionary. LOOP must only be used within a definition. Some compiler security is provided by checking for a 3 on the top of the stack. Since DO leaves a 3 on the stack at compile time (Sequence 2), an un-matched DO and LOOP will probably not have a 3 on the top of the stack and the error will be detected.

The apparent run time action (Sequence 3) of LOOP is to increment the loop Index by one and compare this new Index value with the Limit value. NOTE: This action is actually performed by (LOOP) at run time. If the Index value is less than the Limit value, a branch back to the "loop body" of the DO-LOOP structure is executed. If the Index is equal to or greater than the Limit, then both of these values are dropped from the return stack and interpretation continues with the definition following LOOP .

DO - LOOP structures must be used within a colon definition.

Note that LOOP is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

If an increment value different than 1 is needed, +LOOP should be used.

VLIST is an example of a word which uses LOOP .

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit single precision value 3 used to provide compiler security. (The value 3 is left on the stack at compile time by DO .) The second stack entry contains a 16-bit address specifying the entry point of the DO portion (i.e., the beginning of the "loop body") of the DO-LOOP structure.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - (LOOP) uses no input values from the parameter stack but does expect the Index and Limit values to be on the return stack.
* **At exit** - (LOOP) drops both the Index and Limit values from the return stack when the DO-LOOP structure is exited. (Refer to (LOOP) .)

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

LOOP is a high level colon definition.

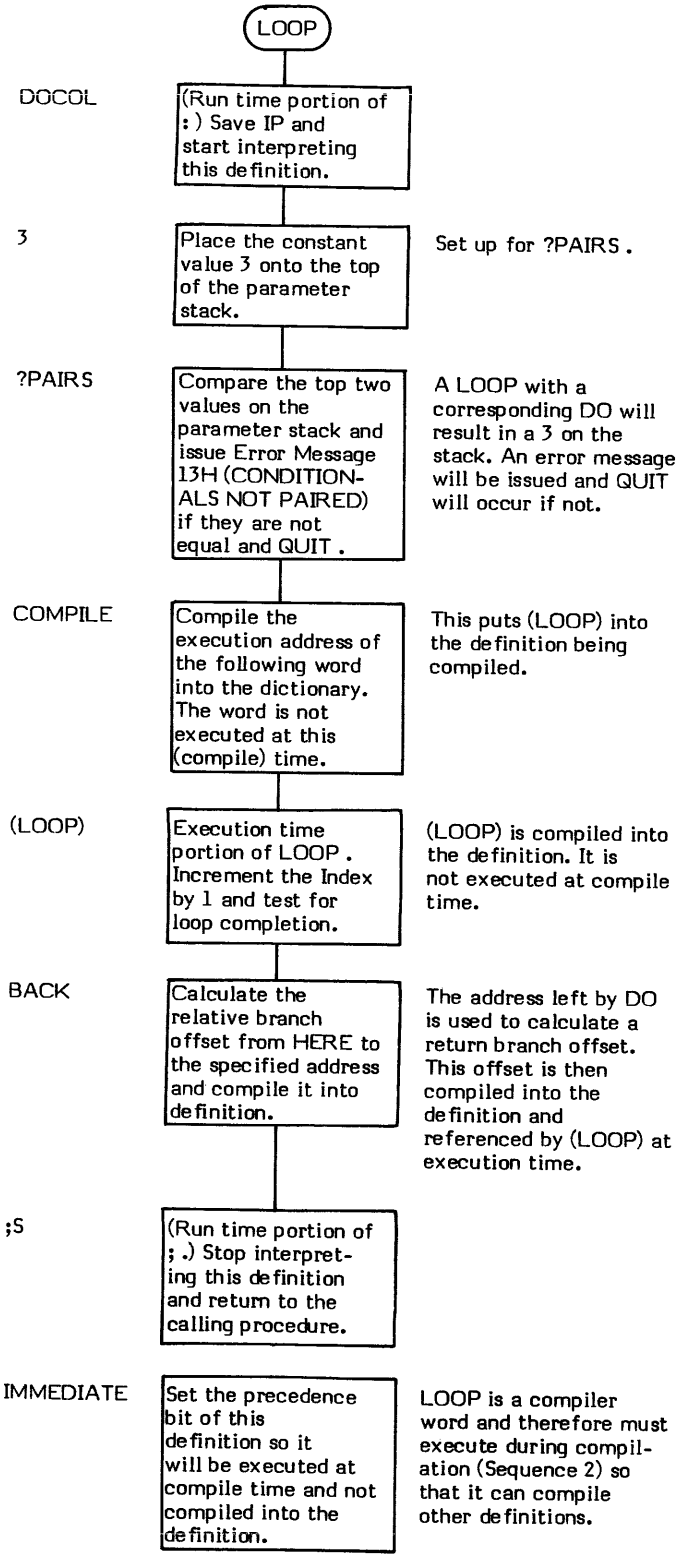
Refer to (LOOP) , DO , +LOOP , and (DO) .

FORTH-79: The FORTH-79 equivalent for LOOP is LOOP .

Definition:

```
:  LOOP  ( Loop address\3 -- )  ( compile time )
       3 ?PAIRS  COMPILE (LOOP)  BACK  ;  IMMEDIATE
```

COMPILE TIME action of LOOP (Sequence 2): (loop address\ 3 --)



EXECUTION TIME action of LOOP (Sequence 3): (--)

The execution time of LOOP is actually that of (LOOP) .

M* (value 1 \ value 2 -- double product)

M* (pronounced "M-star") multiplies two signed single precision values and replaces them with their signed double precision product.

The heart of M* is U* (unsigned multiply). The first part of M* determines the eventual sign of the product, then the unsigned multiplication of U* is performed, and lastly the sign is applied to the product.

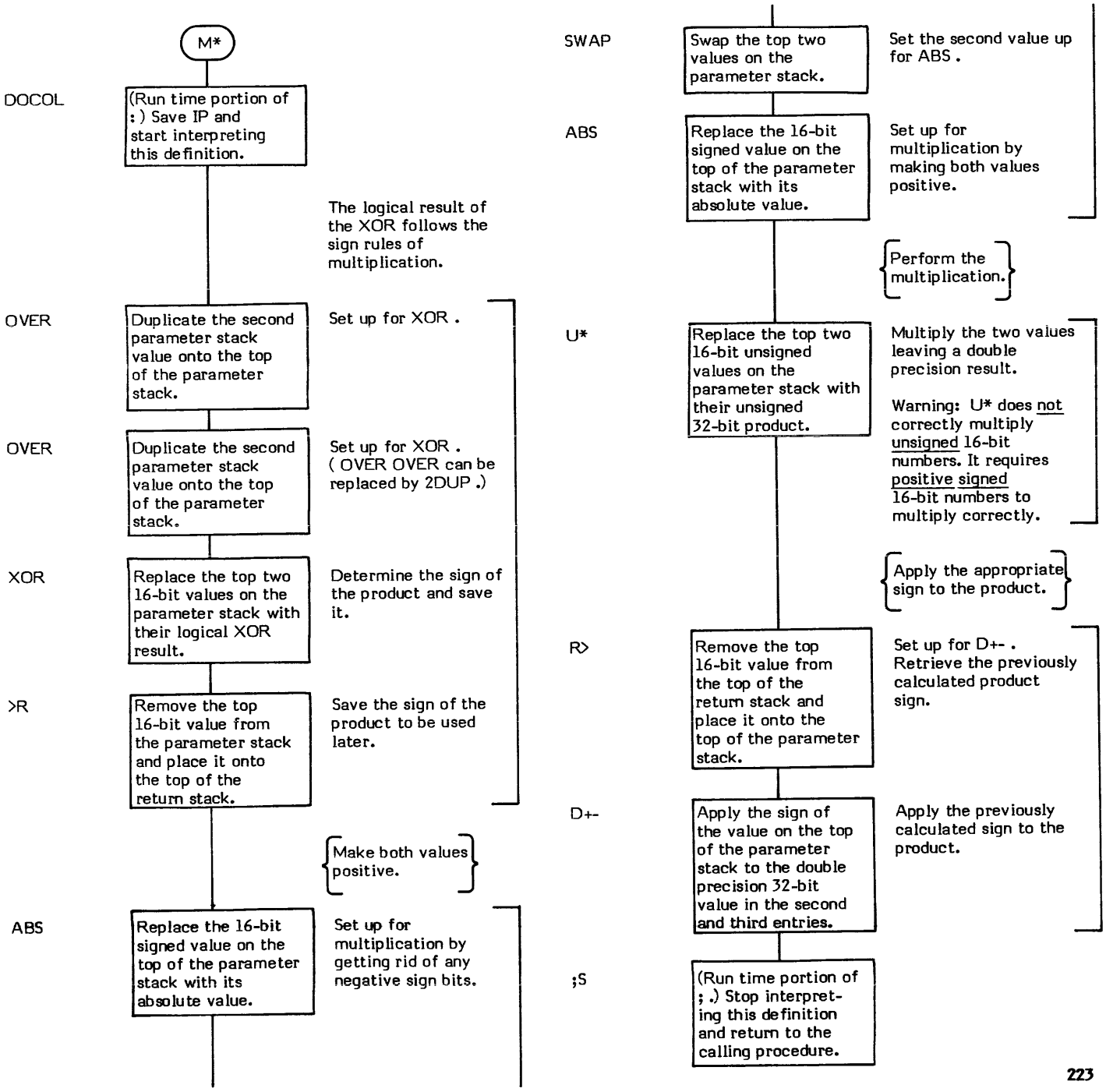
* **At entry** - The top two entries on the parameter stack contain signed 16-bit values to be multiplied together.
* **At exit** - The top two entries of the parameter stack contain a 32-bit signed double precision product. The top entry contains the signed high order portion of the product. The second stack entry contains the low order portion of the product.

M* is a high level colon definition.

Refer to U* .

FORTH-79: There is no FORTH-79 equivalent for M* .

Definition: : M* (value 1 \ value 2 \ value 2 -- double produce)
OVER OVER XOR >R ABS SWAP ABS U* R> D+- ;



M/

M/ (double low \ double high \ divisor -- remainder \ quotient)

M/ (pronounced "M-slash") divides a 32-bit signed double precision value by a 16-bit signed single precision value and replaces them with a 16-bit signed remainder and a 16-bit signed quotient.

The remainder takes its sign from the dividend.

Note that M/ uses a double precision dividend while /MOD uses a single precision dividend.

The heart of M/ is U/ (unsigned divide). The first part of M/ saves the signs of the dividend and divisor, then makes the inputs to U/ positive. After the division, the signs of the quotient and remainder are determined.

* **At entry** - The top of the parameter stack contains a 16-bit signed single precision divisor. The second and third stack entries contain a 32-bit signed double precision dividend. The second stack entry contains the high order portion of the dividend; the third, the low order portion.

* **At exit** - The top of the parameter stack contains the 16-bit signed single precision quotient. The second stack entry contains the 16-bit signed single precision remainder.

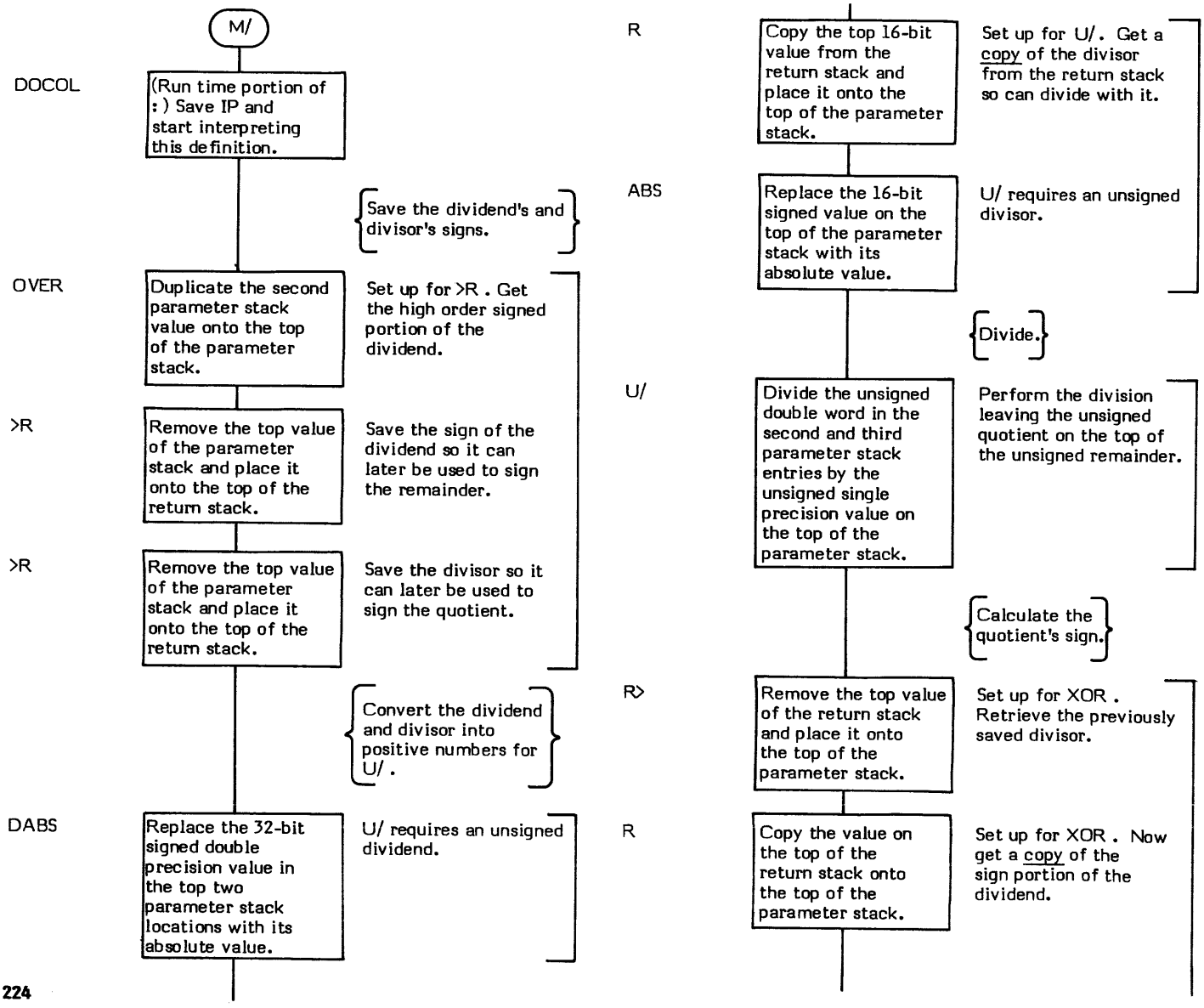
M/ is a high level colon definition.

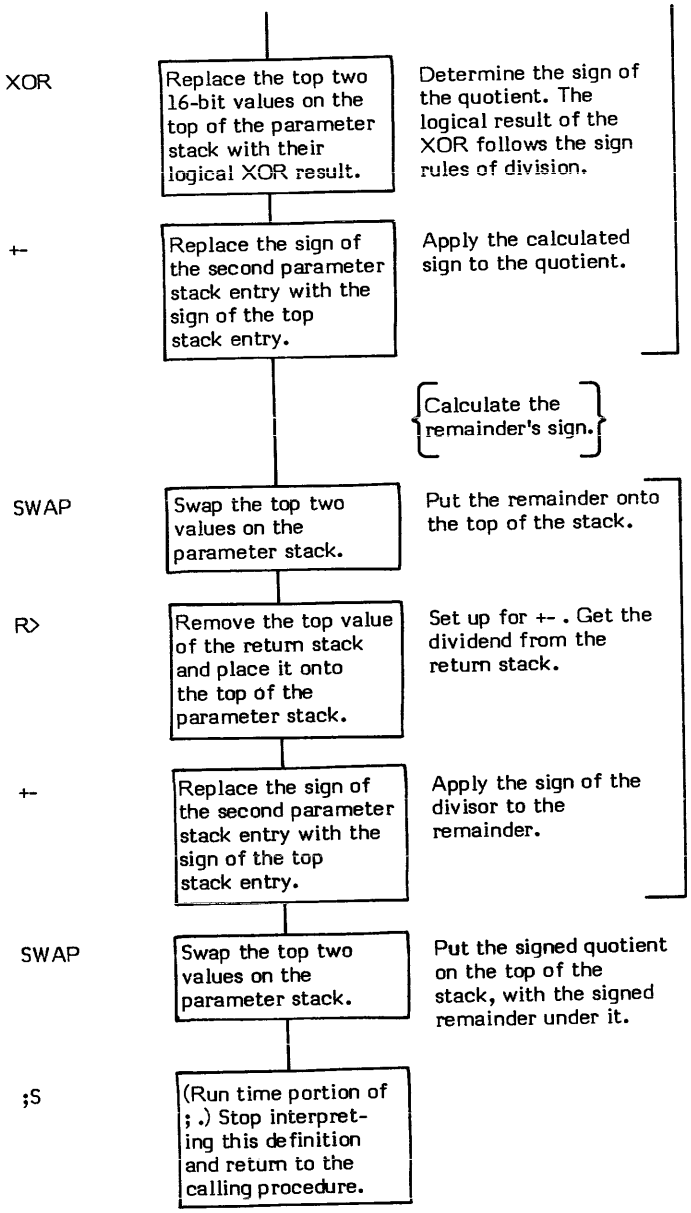
Refer to U/ .

FORTH-79: There is no FORTH-79 equivalent for M/ .

Definition:

```
:  M/   ( double low \ double high \ divisor -- remainder \ quotient )
       OVER >R >R   DABS   R ABS   U/   R> R XOR +-
       SWAP R> +-   SWAP   ;
```




XOR
Replace the top two 16-bit values on the top of the parameter stack with their logical XOR result.
Determine the sign of the quotient. The logical result of the XOR follows the sign rules of division.
+-
Replace the sign of the second parameter stack entry with the sign of the top stack entry.
Apply the calculated sign to the quotient.
Calculate the remainder's sign.
SWAP
Swap the top two values on the parameter stack.
Put the remainder onto the top of the stack.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Set up for +- . Get the dividend from the return stack.
+-
Replace the sign of the second parameter stack entry with the sign of the top stack entry.
Apply the sign of the divisor to the remainder.
SWAP
Swap the top two values on the parameter stack.
Put the signed quotient on the top of the stack, with the signed remainder under it.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

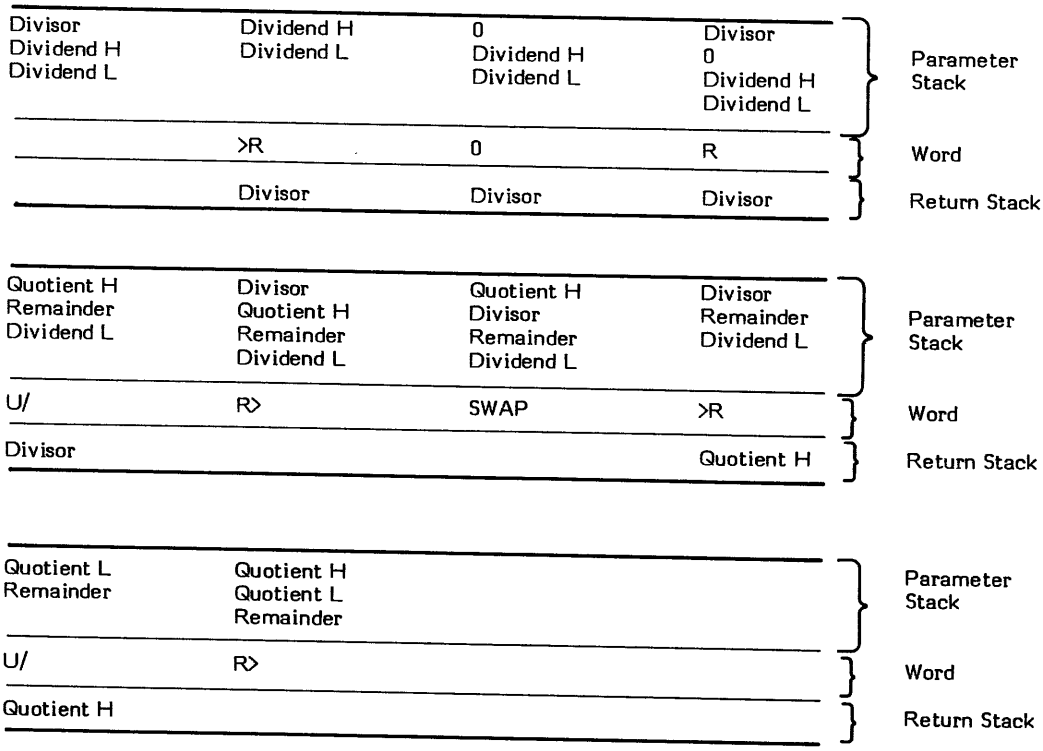
M/MOD

M/MOD (double dividend \ divisor — remainder \ double quotient)

M/MOD (pronounced "M-slash-mod") divides an unsigned double precision value by an unsigned single precision value and replaces them with the unsigned single precision remainder and the unsigned double precision quotient.

U/ is the basis for M/MOD; but, since the output of U/ is only a single precision quotient, some special operations are necessary. In this case, the problem is solved in a manner similar to performing long division by hand. i.e., The high order portion of the dividend is divided, rendering half of the quotient and a remainder. This remainder and the low order dividend are again divided, rendering the rest of the double precision quotient and a remainder.

The stack activity of M/MOD is confusing enough that a step by step description of the effect of each word is necessary to properly understand its operation:



* **At entry** - The top of the parameter stack contains a 16-bit unsigned single precision divisior. The second and third stack entries contain a 32-bit unsigned double precision dividend, with the high order portion of the dividend in the second stack entry and the low order in the third.

* **At exit** - The top two entries of the parameter stack contain a 32-bit unsigned double precision quotient. The top entry contains the high order portion of the quotient. The second stack entry contains the low order portion of the quotient. The third stack entry contains the 16-bit unsigned single precision remainder.

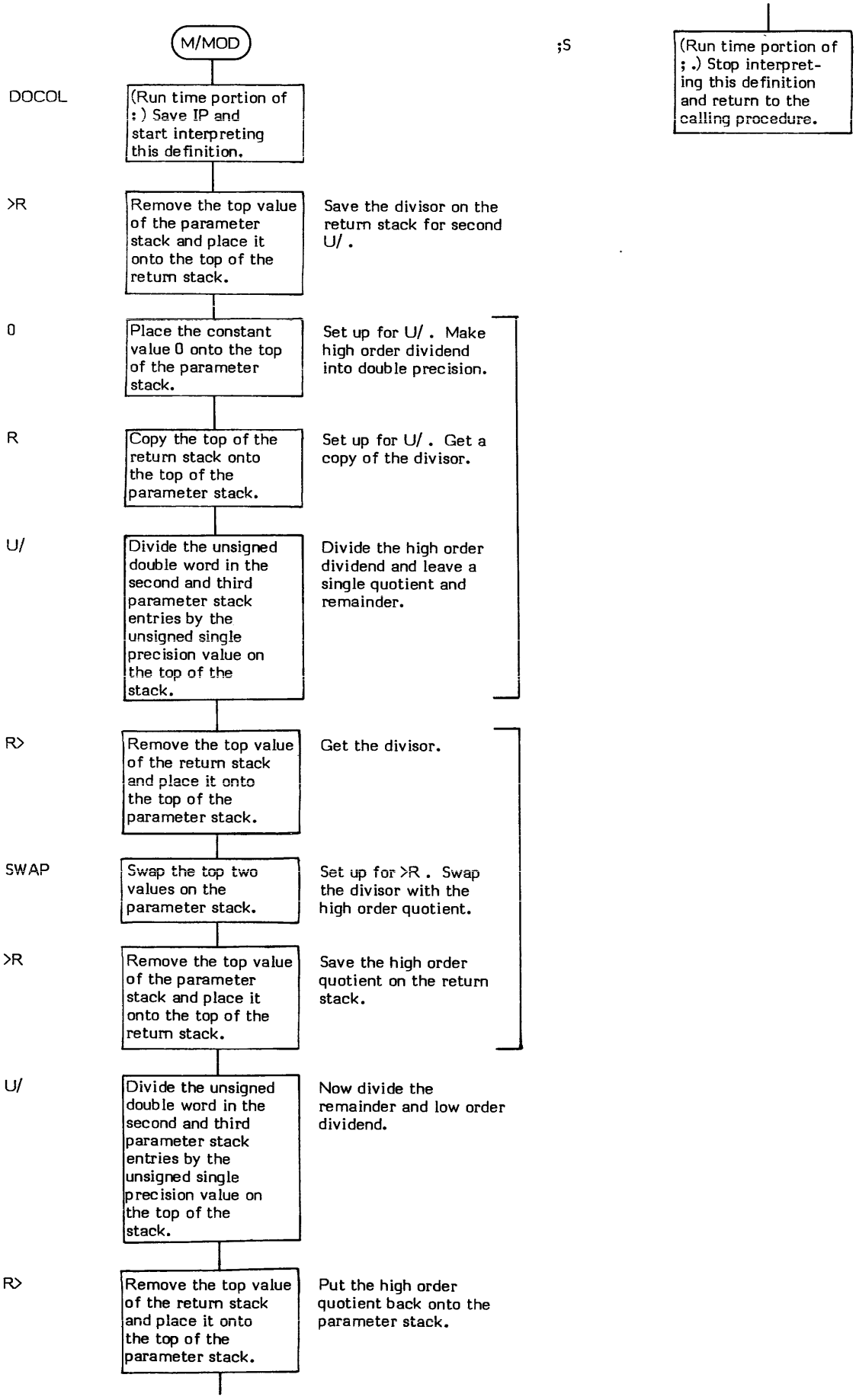
M/MOD is a high level colon definition.

Refer to U/ .

FORTH-79: There is no FORTH-79 equivalent for M/MOD .

Definition:

```
:  M/MOD  ( double dividend \ divisor -- remainder \ double quotient )
     >R  0 R U/  R> SWAP >R  U/  R>  ;
```


M/MOD
DOCOL
(Run time portion of :) Save IP and start interpreting this definition.
>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Save the divisor on the return stack for second U/ .
0
Place the constant value 0 onto the top of the parameter stack.
Set up for U/ . Make high order dividend into double precision.
R
Copy the top of the return stack onto the top of the parameter stack.
Set up for U/ . Get a copy of the divisor.
U/
Divide the unsigned double word in the second and third parameter stack entries by the unsigned single precision value on the top of the stack.
Divide the high order dividend and leave a single quotient and remainder.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Get the divisor.
SWAP
Swap the top two values on the parameter stack.
Set up for >R . Swap the divisor with the high order quotient.
>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Save the high order quotient on the return stack.
U/
Divide the unsigned double word in the second and third parameter stack entries by the unsigned single precision value on the top of the stack.
Now divide the remainder and low order dividend.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Put the high order quotient back onto the parameter stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

MAX

MAX (value \value -- maximum value)

MAX (pronounced "max") compares the top two 16-bit signed values on the parameter stack, drops the smaller of the two values, and leaves the larger (or MAXimum).

MIN performs the opposite function of MAX .

SPACES is an example of a word which uses MAX .

* **At entry** - The top of the parameter stack contains one of the 16-bit signed single precision values to be compared. The second stack entry contains the other 16-bit signed single precision value to be compared.

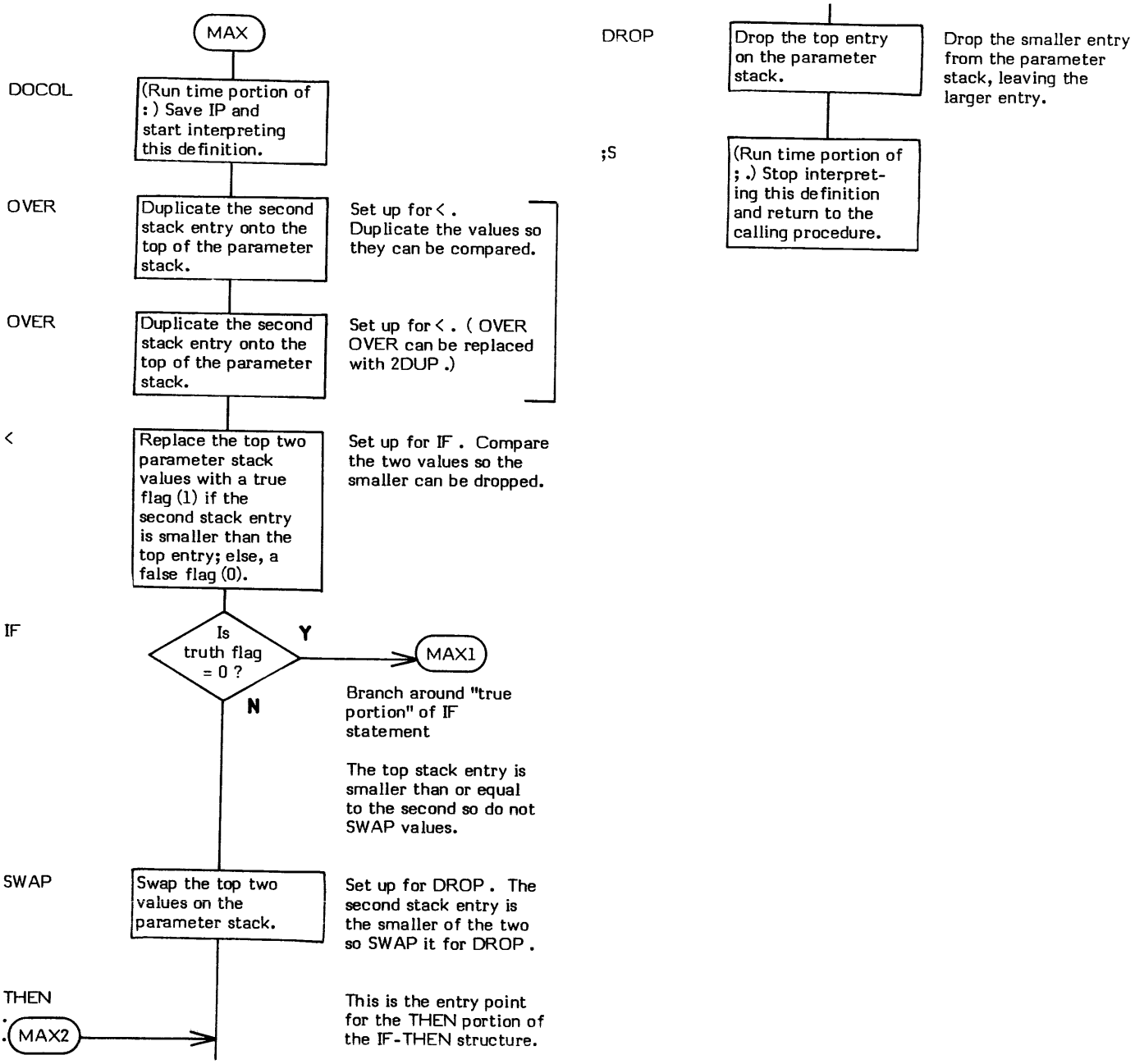
* **At exit** - The top of the parameter stack contains the larger of the two 16-bit values.

MAX is a high level colon definition.

Refer to MIN .

FORTH-79: The FORTH-79 equivalent for MAX is MAX .

Definition: : MAX (value\ value -- maximum value)
OVER OVER < IF SWAP THEN DROP ;



MESSAGE (message number —)

MESSAGE outputs a selected message to the output device. The message can either be just a message number (if the contents of the user variable WARNING is a 0) or it is a line ot text (if the contents of WARNING is a non-zero value).

The message number is supplied as a signed single precision value. When WARNING contains a 1, this number is used as a line offset relative to Line 0 of Screen 4 in Drive 0. That is to say that a message number of +2 will cause Line 2 on Screen 4 to be output. A value of +19 will cause Line 3 of Screen 5 to be output. A value of -1 will cause Line 16 of Screen 3 to be output.

Message Number 0 will not output anything as Line 0 of Screen 4 is a comment. Note that an Error Message Number 0 input to ERROR causes a ? ("HUH?") to be printed.

All relative message text lines are based on Line 0 of Screen 4 in Drive 0. MESSAGE adjusts its input to .LINE by what is in OFFSET so all references are always relative to Drive 0.

Note that, while the FORTH system uses MESSAGE for error messages, its use is not restricted to that purpose. Applications may use the word to output any manner of desired messages stored anywhere on disk.

It is a good idea to always reserve the error message screens for error messages on all disks. e.g., A data disk that has a data file in the location where error messages are kept will cause undetermined garbage to be displayed if an error occurs and MESSAGE outputs a message from what it "considers" to be valid error message text.

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision message number.
* **At exit** - No parameters.

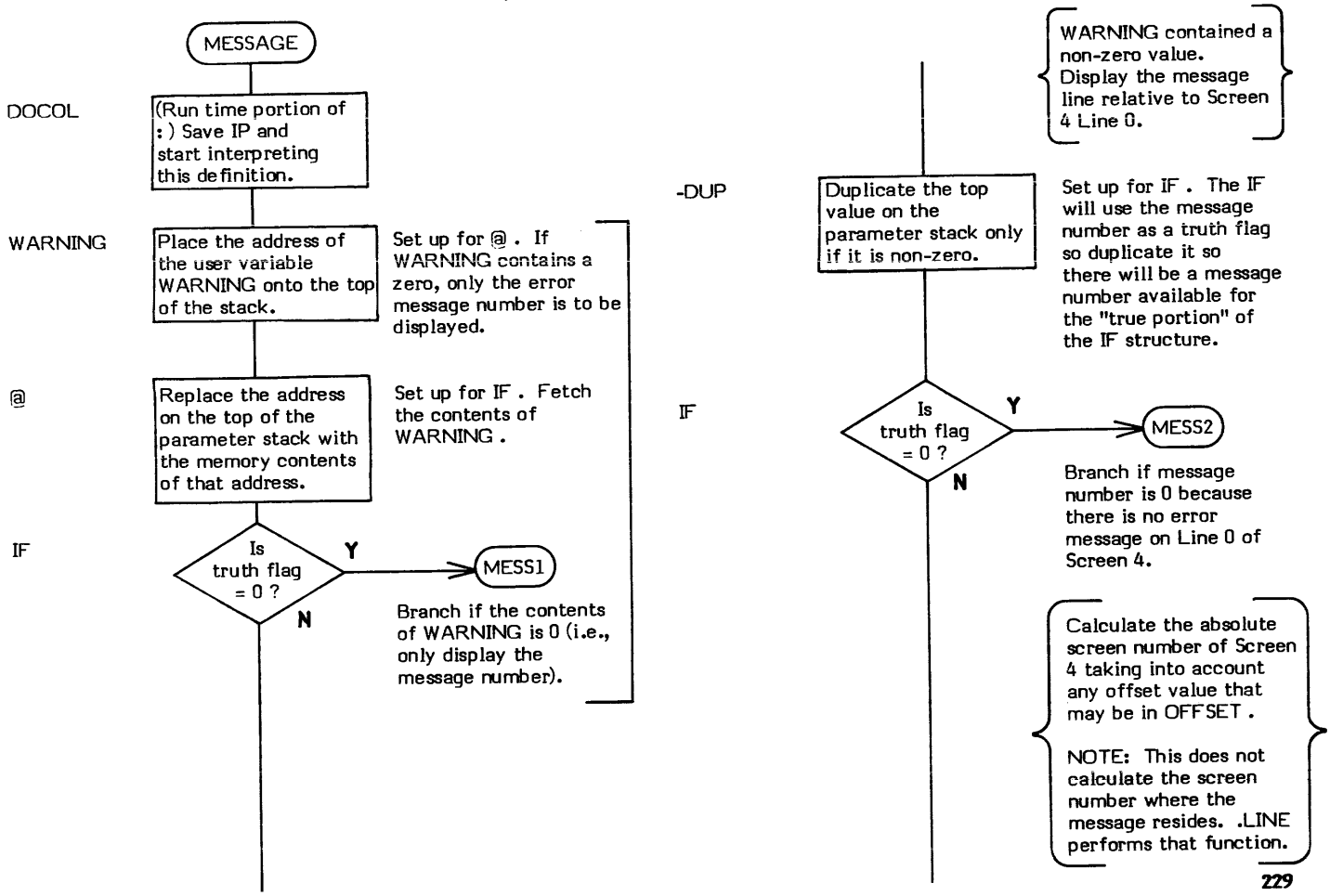
MESSAGE is a high level colon definition.

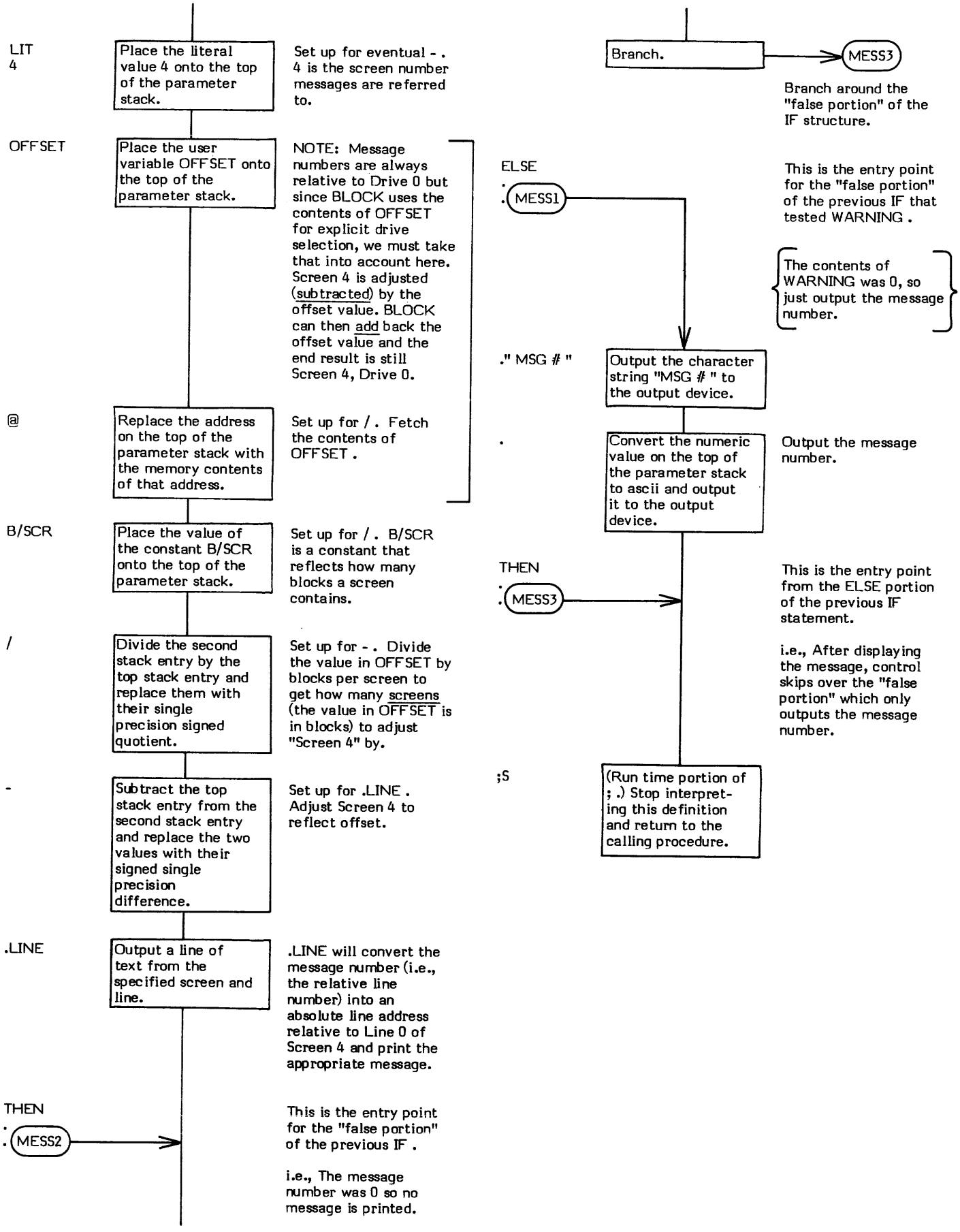
Refer to WARNING , ERROR , .LINE , OFFSET , and BLOCK .

FORTH-79: There is no FORTH-79 equivalent for MESSAGE .

Definition:

```
:  MESSAGE  ( # -- )
     WARNING @  IF
                   -DUP  IF
                            4  OFFSET @  B/SCR / - .LINE
                         THEN
     ELSE  ." MSG " .
     THEN  ;
```




LIT
4
Place the literal value 4 onto the top of the parameter stack.
Set up for eventual - . 4 is the screen number messages are referred to.
OFFSET
Place the user variable OFFSET onto the top of the parameter stack.
NOTE: Message numbers are always relative to Drive 0 but since BLOCK uses the contents of OFFSET for explicit drive selection, we must take that into account here. Screen 4 is adjusted (subtracted) by the offset value. BLOCK can then add back the offset value and the end result is still Screen 4, Drive 0.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for / . Fetch the contents of OFFSET .
B/SCR
Place the value of the constant B/SCR onto the top of the parameter stack.
Set up for / . B/SCR is a constant that reflects how many blocks a screen contains.
/
Divide the second stack entry by the top stack entry and replace them with their single precision signed quotient.
Set up for - . Divide the value in OFFSET by blocks per screen to get how many screens (the value in OFFSET is in blocks) to adjust "Screen 4" by.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed single precision difference.
Set up for .LINE . Adjust Screen 4 to reflect offset.
.LINE
Output a line of text from the specified screen and line.
.LINE will convert the message number (i.e., the relative line number) into an absolute line address relative to Line 0 of Screen 4 and print the appropriate message.
THEN
. MESS2
This is the entry point for the "false portion" of the previous IF .
i.e., The message number was 0 so no message is printed.
Branch.
MESS3
Branch around the "false portion" of the IF structure.
ELSE
. MESS1
This is the entry point for the "false portion" of the previous IF that tested WARNING .
The contents of WARNING was 0, so just output the message number.
." MSG # "
Output the character string "MSG # " to the output device.
.
Convert the numeric value on the top of the parameter stack to ascii and output it to the output device.
Output the message number.
THEN
. MESS3
This is the entry point from the ELSE portion of the previous IF statement.
i.e., After displaying the message, control skips over the "false portion" which only outputs the message number.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

MIN (value\value -- minimum value)

MIN (pronounced "min") compares the top two 16-bit signed values on the parameter stack, drops the larger of the two values, and leaves the smaller (or MINimum).

MAX performs the opposite function of MIN .

CREATE is an example of a word what uses MIN .

* **At entry** - The top of the parameter stack contains one of the 16-bit signed single precision values to be compared. The second stack entry contains the other 16-bit signed single precision value to be compared.

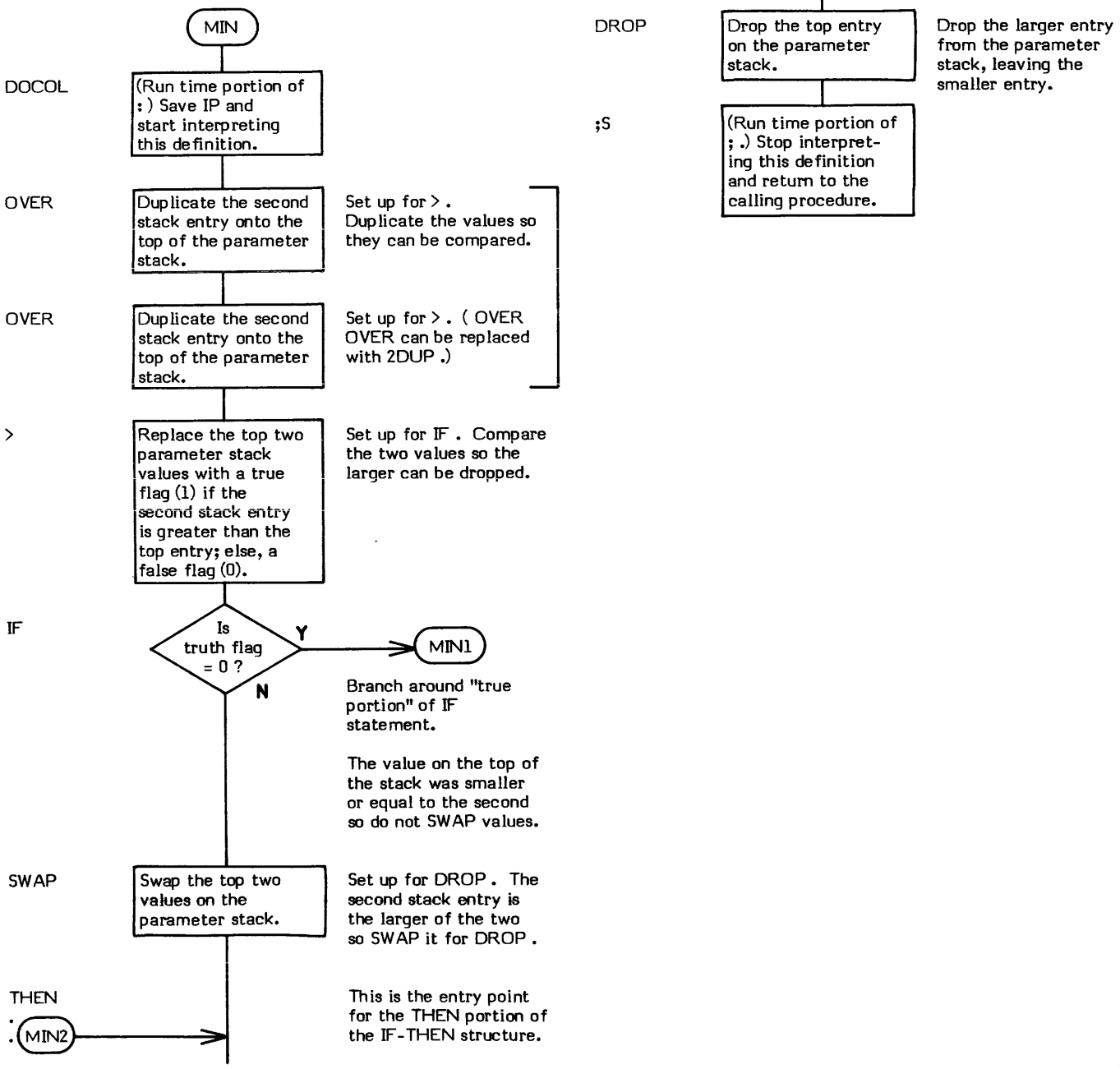
* **At exit** - The top of the parameter stack contains the smaller of the two 16-bit signed values.

MIN is a high level colon definition.

Refer to MAX .

FORTH-79: The FORTH-79 equivalent for MIN is MIN .

Definition: : MIN (value\value -- minimum value)
OVER OVER > IF SWAP THEN DROP ;



MINUS

MINUS (value to be negated — two's complement)

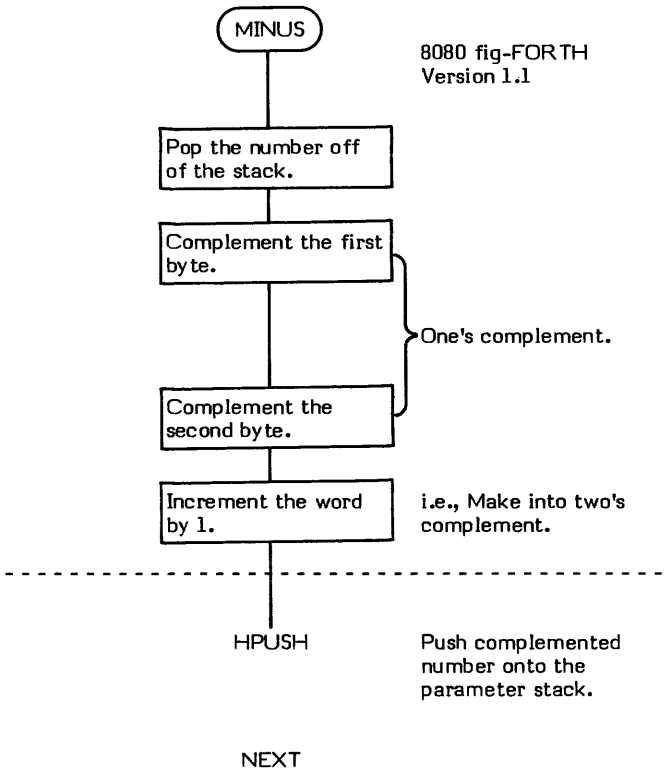
MINUS replaces the 16-bit value on the top of the parameter stack with its two's complement.

- is an example of a word which uses MINUS .

* **At entry** - The top of the parameter stack contains the 16-bit single precision value to be converted.
* **At exit** - The top of the parameter stack contains the 16-bit single precision two's complement of the given value.

MINUS is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for MINUS is NEGATE .



MOD (dividend \ divisor — remainder)

MOD (pronounced "mod") divides a 16-bit signed single precision value by another 16-bit signed single precision value and replaces them with their 16-bit signed remainder (or modulo, hence the name MOD). The remainder takes its sign from the dividend.

The basis of MOD is /MOD . /MOD leaves a quotient and remainder. MOD drops the quotient. This is very similar to / , which drops the remainder.

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision divisor. The second stack entry contains the 16-bit signed single precision dividend.

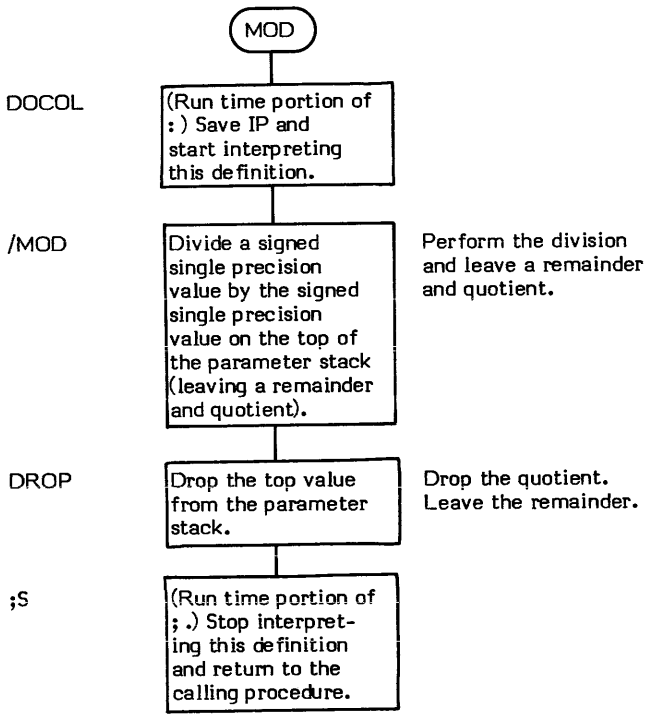
* **At exit** - The top of the parameter stack contains the 16-bit signed single precision remainder.

MOD is a high level colon definition.

Refer to /MOD .

FORTH-79: The FORTH-79 equivalent for MOD is MOD .

Definition:
```
:  MOD  ( dividend \ divisor -- remainder )
     /MOD  DROP  ;
```



MON

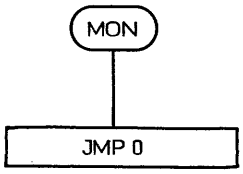
MON (–)

MON (pronounced "mon") exits to the system monitor (hence "MON") of the system FORTH is running on (or under). The word may have other names (e.g., the 8080 1.1 CP/M Version uses BYE) but the function is always the same.

* **At entry** - No parameters.
* **At exit** - No parameters.

MON may be a high or low level definition depending upon the specific installation.

FORTH-79: There is no FORTH-79 equivalent for MON .



8080 fig-FORTH
Version 1.1

Jump to the CP/M warm
start vector.

NEXT

NEXT is the FORTH inner interpreter. (Actually NEXT is the entry point label of the inner interpreter procedure.) This procedure is not a part of the FORTH dictionary. Its purpose is to sequentially execute the "next" Code Field Address contained within the Parameter Field of a compiled definition.

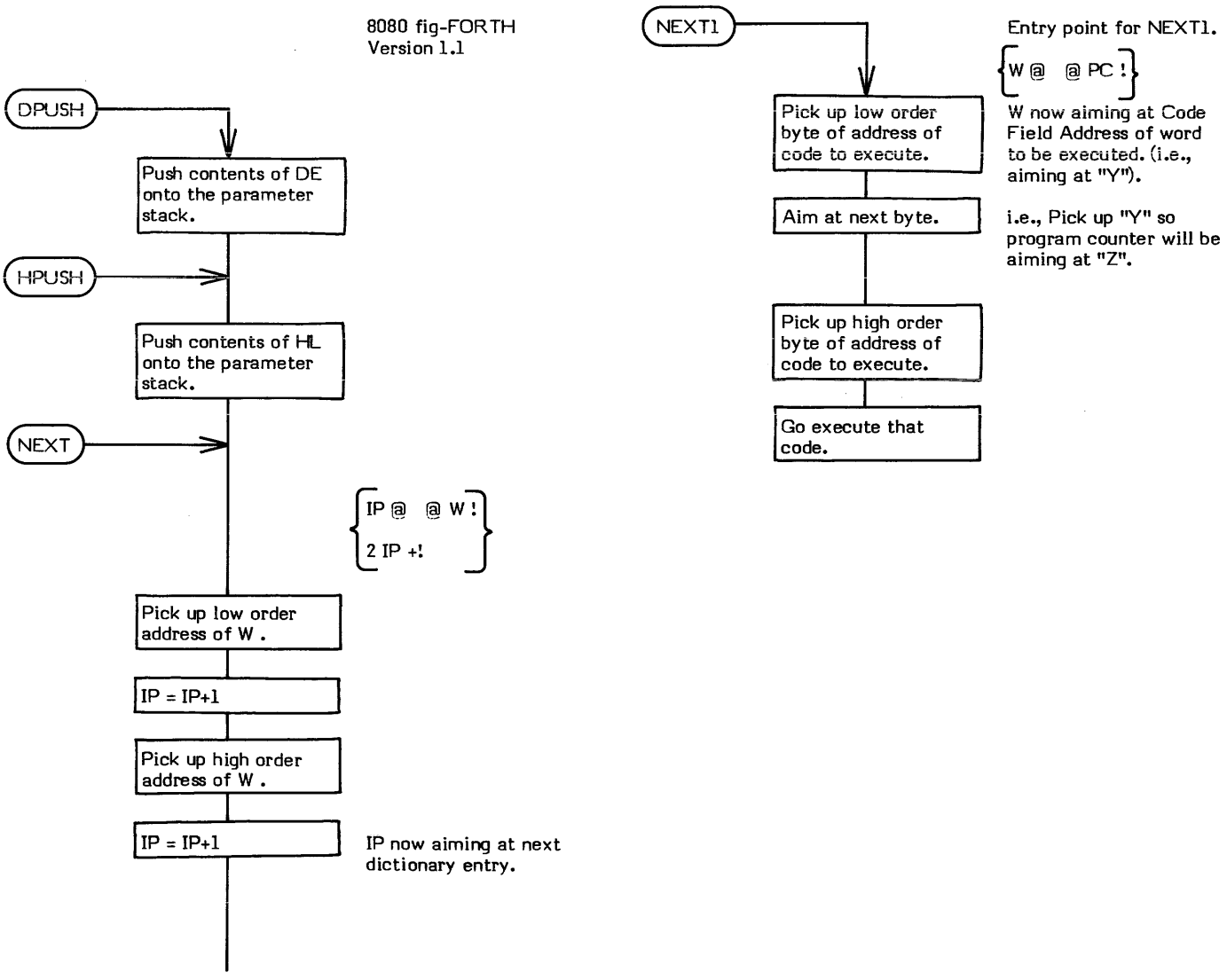
NEXT only sequentially "executes Code Field Addresses"; it does not of itself perform any nesting or un-nesting. Nesting is performed via a word such as : (DOCOL actually); un-nesting is performed via a word such as ;S .

All definitions must eventually be terminated with a jump to NEXT . Code definitions must end by directly jumping to NEXT or by performing the same function as NEXT (e.g., EXECUTE).

The action of NEXT can be symbolically described in high level terms as follows:

IP @	Fetch the contents of IP . (IP points to the next CFA , i.e., "word", to execute.)
@ W !	Fetch the CFA IP is pointing to and store it into W .
2 IP +!	Increment IP to point to the next "word" to execute.
W @	Fetch the contents of W . (W now points to the Code Field of the defintion to be executed.)
@ PC !	Fetch the contents of this definition's Code Field (i.e., the address to execute) and put it into the CPU's program counter.

Refer to W , IP , : , ;S , and DOES> .



NFA

NFA (Parameter Field Address -- Name Field Address)

NFA (pronounced "N-F-A") converts a given Parameter Field Address (PFA) of a dictionary definition into its Name Field Address (NFA).

The structure of the header of a FORTH definition is:

Name Field	Variable length
Link Field	2 byte address pointer
Code Field	2 byte address pointer
Parameter Field	Variable length

* **At entry** - The top of the parameter stack contains the 16-bit Parameter Field Address of a FORTH definition.
* **At exit** - The top of the parameter stack contains the 16-bit Name Field Address of the specified FORTH definition.

CREATE is an example of a word that uses NFA .

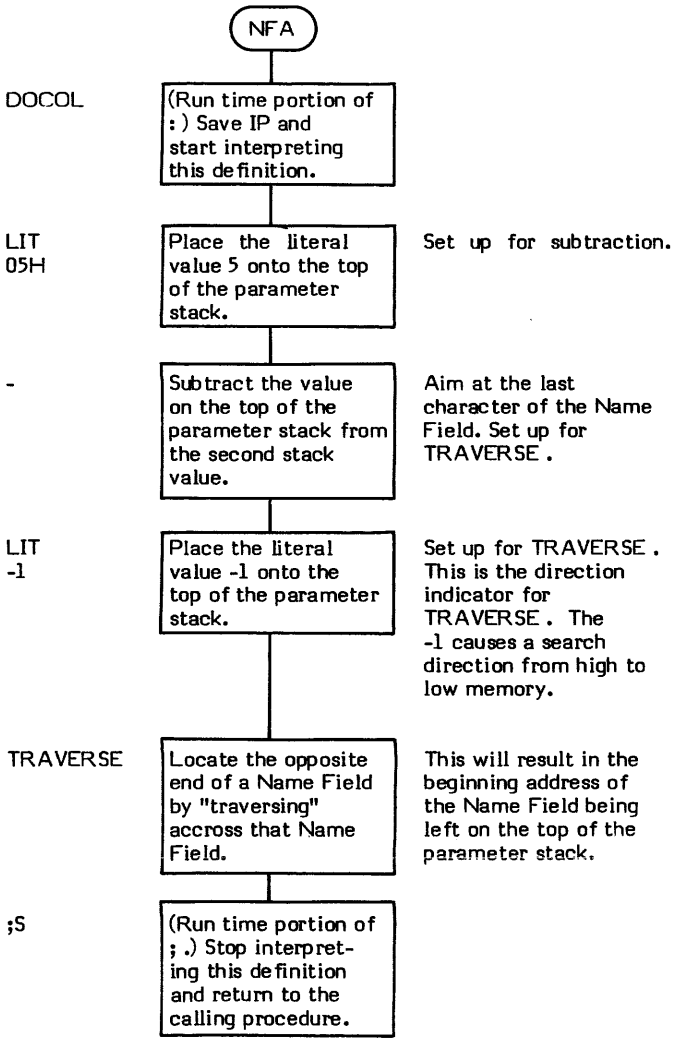
NOTE: Since the fig-FORTH Name Field is a variable length, a simple subtraction cannot be used to directly calculate its beginning address. TRAVERSE must be used instead. The exact nature of NFA may vary from system to system because of differing word sizes, etc.

NFA is a high level colon definition.

Refer to PFA , LFA , and CFA .

FORTH-79: There is no FORTH-79 equivalent for NFA . A FORTH-79 program may not address into a definition's Name Field.

Definition:
```
: NFA ( PFA -- NFA )
    5 -   -1 TRAVERSE   ;
```



NULL (–)

NULL (pronounced "null" and sometimes called "X") is the word that causes an exit from the endless loop in the INTERPRET procedure. "NULL" is a pseudonym for an ascii null character (00). The actual Name Field of NULL consists of one ascii null character.

Despite its rather non-standard name, NULL is a perfectly valid definition. That is to say that as INTERPRET sequentially moves through a buffer, compiling and interpreting as it goes, the end of the buffer will eventually be reached. By convention, all buffers must end with a null character. When INTERPRET encounters this null character, it searches the dictionary for a Name Field match. A match occurs when the null definition is encountered. Since NULL is a valid definition and is also IMMEDIATE , it is executed.

NULL causes an exit from INTERPRET by dropping the top address of the return stack. Then, when NULL returns, it exits not to the level it was at (i.e., back to INTERPRET) but instead exits to one level of nesting higher (i.e., back to the definition that contained INTERPRET -- more specifically to the definition immediately following INTERPRET .) For example, the word STATE in QUIT , or R> in LOAD .

A large portion of the coding in NULL is devoted to checking to ensure that the system is not in compilation state if the null character encountered is at the end of an editing screen. (It must be determined if the current buffer is the last buffer in a screen.) This check is performed because it is illegal to extend compilation across editing screens. Error message 12H (EXECUTION ONLY) is issued if this condition is detected.

Note that NULL is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

EXECUTION ONLY (12H) -- This word must not be used while the system is in compile mode.

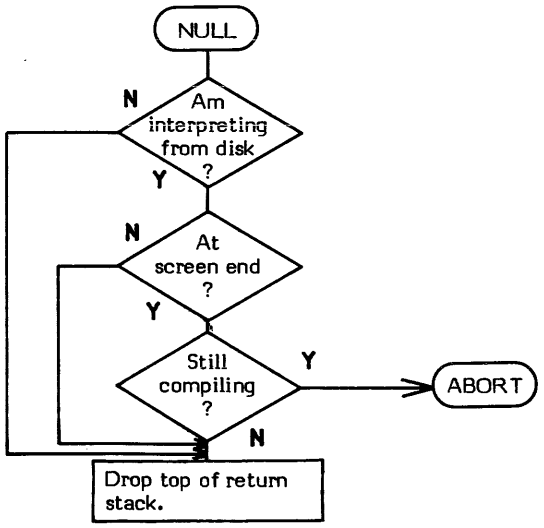
NULL is a high level colon definition.

Refer to INTERPRET , QUIT , and LOAD .

FORTH-79: There is no FORTH-79 equivalent for NULL .

Definition:

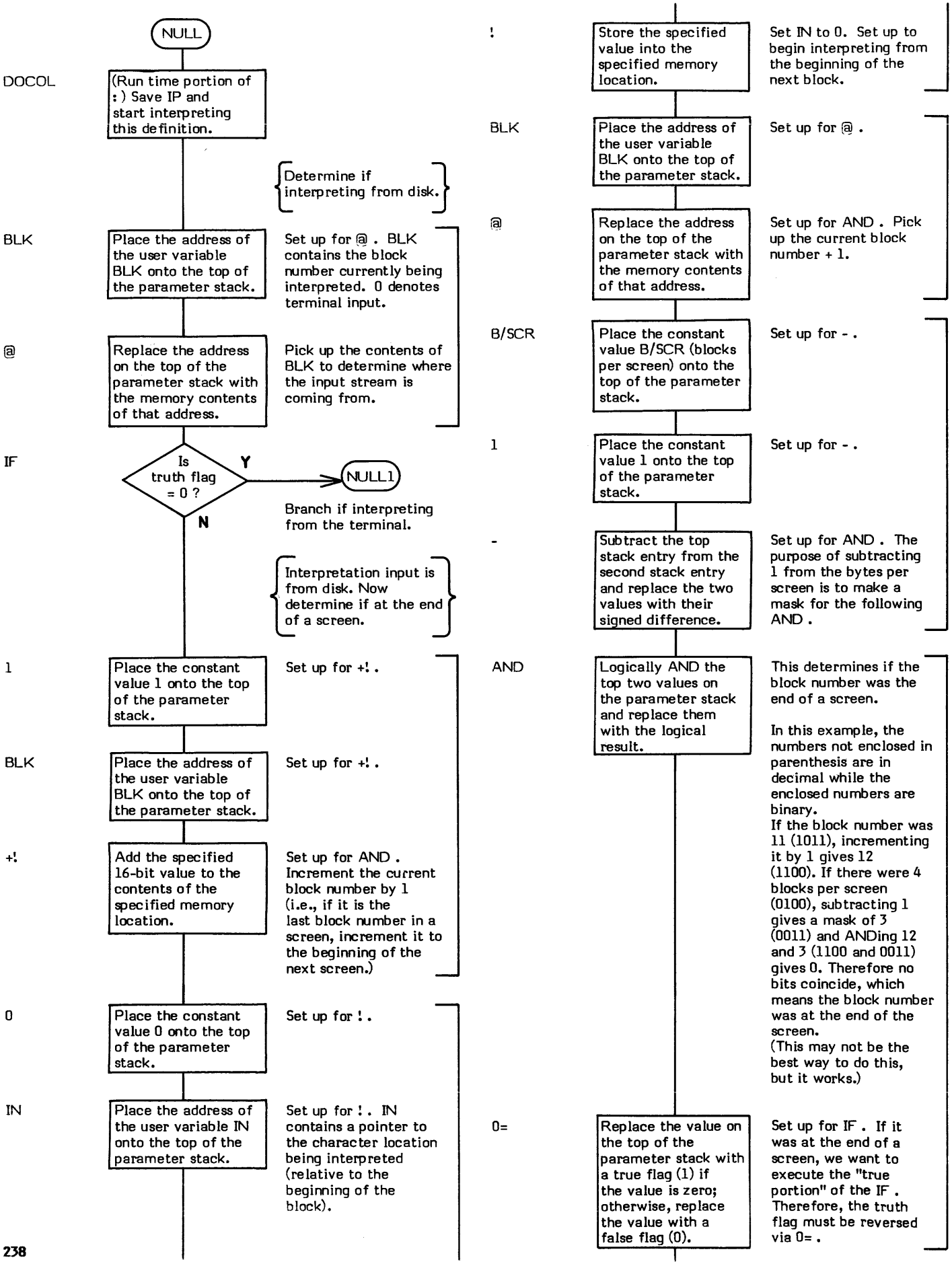
```
:  NULL  ( -- )
     BLK @      IF
          1 BLK +!   0 IN !   BLK @   B/SCR 1 -
          AND   0=   IF   ?EXEC   R>   DROP   THEN
     ELSE   R>   DROP
     THEN   ;   IMMEDIATE
```

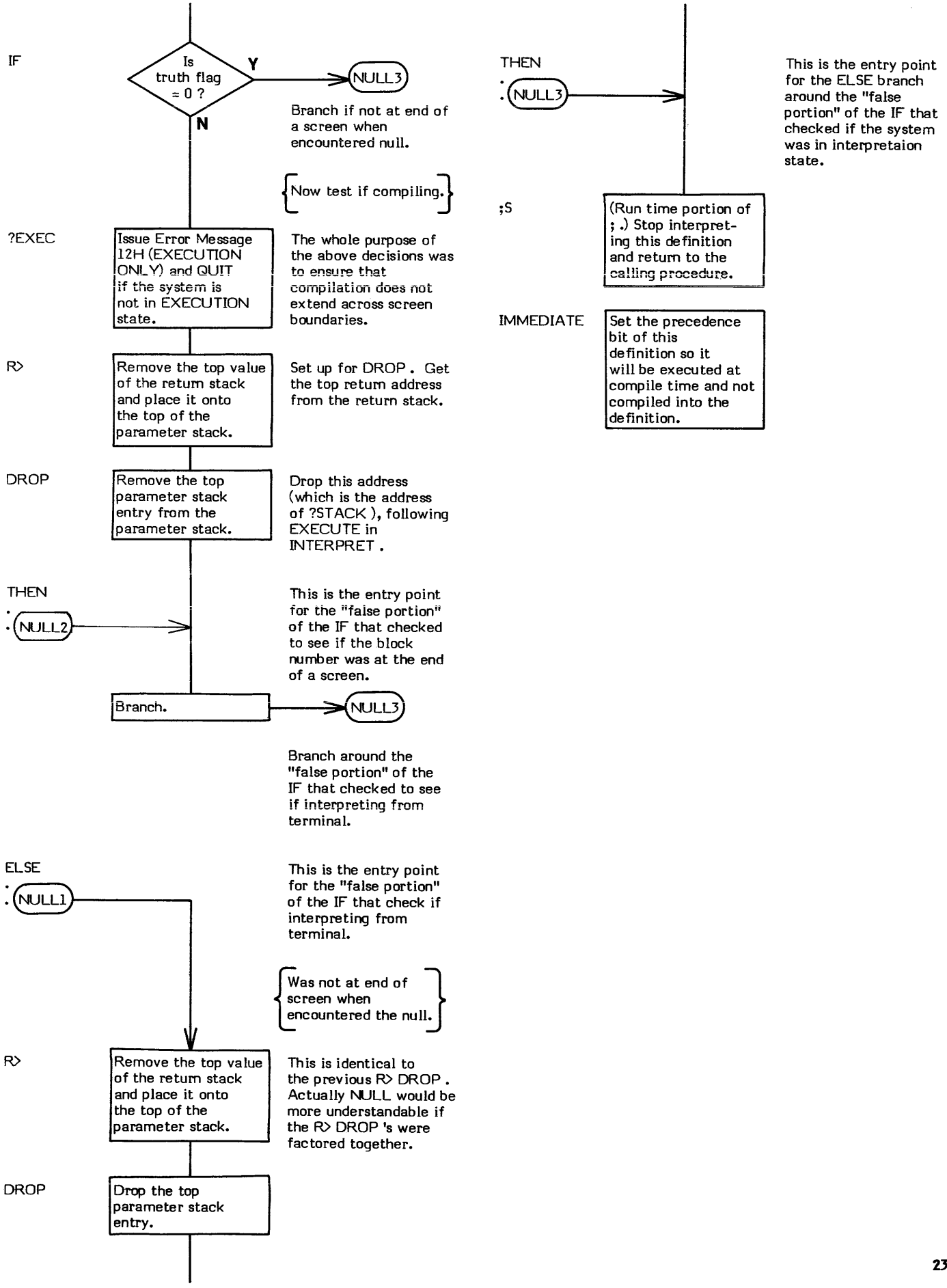



```
;S          Return to next higher
            level. i.e., Level that
            "called" INTERPRET .
```

Figure NULL-1

High Level Flowchart of NULL


NULL
DOCOL
(Run time portion of :) Save IP and start interpreting this definition.
Determine if interpreting from disk.
BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for @ . BLK contains the block number currently being interpreted. 0 denotes terminal input.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the contents of BLK to determine where the input stream is coming from.
IF
Is truth flag = 0 ?
Y
NULL1
N
Branch if interpreting from the terminal.
Interpretation input is from disk. Now determine if at the end of a screen.
1
Place the constant value 1 onto the top of the parameter stack.
Set up for +! .
BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for +! .
+!
Add the specified 16-bit value to the contents of the specified memory location.
Set up for AND . Increment the current block number by 1 (i.e., if it is the last block number in a screen, increment it to the beginning of the next screen.)
0
Place the constant value 0 onto the top of the parameter stack.
Set up for ! .
IN
Place the address of the user variable IN onto the top of the parameter stack.
Set up for ! . IN contains a pointer to the character location being interpreted (relative to the beginning of the block).
!
Store the specified value into the specified memory location.
Set IN to 0. Set up to begin interpreting from the beginning of the next block.
BLK
Place the address of the user variable BLK onto the top of the parameter stack.
Set up for @ .
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for AND . Pick up the current block number + 1.
B/SCR
Place the constant value B/SCR (blocks per screen) onto the top of the parameter stack.
Set up for - .
1
Place the constant value 1 onto the top of the parameter stack.
Set up for - .
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Set up for AND . The purpose of subtracting 1 from the bytes per screen is to make a mask for the following AND .
AND
Logically AND the top two values on the parameter stack and replace them with the logical result.
This determines if the block number was the end of a screen.
In this example, the numbers not enclosed in parenthesis are in decimal while the enclosed numbers are binary.
If the block number was 11 (1011), incrementing it by 1 gives 12 (1100). If there were 4 blocks per screen (0100), subtracting 1 gives a mask of 3 (0011) and ANDing 12 and 3 (1100 and 0011) gives 0. Therefore no bits coincide, which means the block number was at the end of the screen.
(This may not be the best way to do this, but it works.)
0=
Replace the value on the top of the parameter stack with a true flag (1) if the value is zero; otherwise, replace the value with a false flag (0).
Set up for IF . If it was at the end of a screen, we want to execute the "true portion" of the IF . Therefore, the truth flag must be reversed via 0= .


IF
Is truth flag = 0 ?
Y
N
NULL3
Branch if not at end of a screen when encountered null.
Now test if compiling.
?EXEC
Issue Error Message 12H (EXECUTION ONLY) and QUIT if the system is not in EXECUTION state.
The whole purpose of the above decisions was to ensure that compilation does not extend across screen boundaries.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Set up for DROP . Get the top return address from the return stack.
DROP
Remove the top parameter stack entry from the parameter stack.
Drop this address (which is the address of ?STACK), following EXECUTE in INTERPRET .
THEN
NULL2
This is the entry point for the "false portion" of the IF that checked to see if the block number was at the end of a screen.
Branch.
NULL3
Branch around the "false portion" of the IF that checked to see if interpreting from terminal.
ELSE
NULL1
This is the entry point for the "false portion" of the IF that check if interpreting from terminal.
Was not at end of screen when encountered the null.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
This is identical to the previous R> DROP . Actually NULL would be more understandable if the R> DROP 's were factored together.
DROP
Drop the top parameter stack entry.
THEN
NULL3
This is the entry point for the ELSE branch around the "false portion" of the IF that checked if the system was in interpretaion state.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of this definition so it will be executed at compile time and not compiled into the definition.

NUMBER

NUMBER (address of text string — double precision value)

NUMBER uses the current base to convert a character string into a signed double precision number. The character string must begin with a length byte (e.g., standard "WORD" format). The position of the last decimal point encountered (if any are present) will be left in DPL .

Error Message 0 ("?") is issued if a non-convertible character is encountered.

The ascii character blank (20H) terminates the conversion process.

Note that there are only two valid non-convertible characters:

1. Decimal points, that are effectively ignored.

2. A minus sign in the first character, that is used to set the sign bit to negative.

A high level logic flowchart of NUMBER is provided as Figure NUMBER-1 so that the overall logic of the word can be more easily grasped.

INTERPRET is an example of a word that uses NUMBER .

* **At entry** - The top of the parameter stack contains a 16-bit address which points to the text string to be converted. The first byte of the text string contains the length of the following string. A decimal point may be present anywhere in the text. The first character of the text string may be a minus sign signifying a negative number. The user variable BASE contains the numeric conversion base.

* **At exit** - The signed 32-bit double precision numeric conversion value is on the top of the parameter stack with the most significant word on the top of the stack and the least significant word as the second entry. The user variable DPL contains a number reflecting the number of digits occurring to the right of the last encountered decimal point. A value of -1 indicates no decimal point was encountered.

LIKELY ERROR MESSAGES:

? pronounced "HUH?" (0) -- The word in question is not a number.

NUMBER is a high level colon definition.

Refer to (NUMBER) , DPL , BASE , and WORD .

FORTH-79: The FORTH-79 equivalent for NUMBER is CONVERT .

Definition:

```
:  NUMBER    ( address of text string -- double value )
      0 0 ROT    DUP 1+ C@    2D =    DUP >R
      +    -1          BEGIN
                           DPL !    (NUMBER)    DUP C@    BL -
                       WHILE
                           DUP C@    2E - 0 ?ERROR    0
                       REPEAT
      DROP    R>    IF    DMINUS    THEN    ;
```

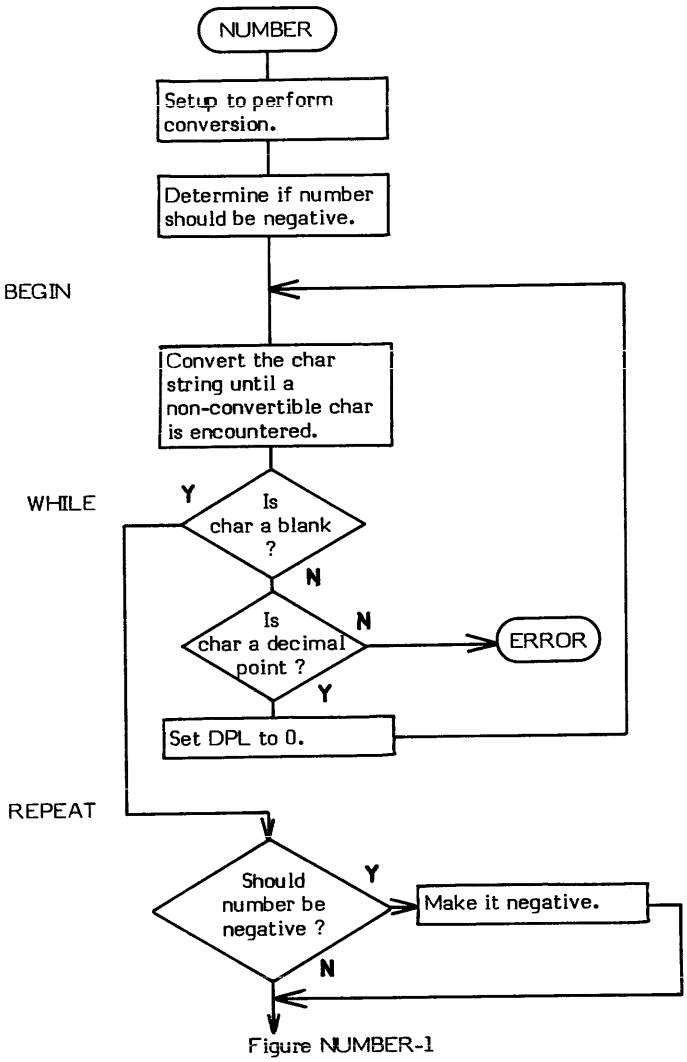
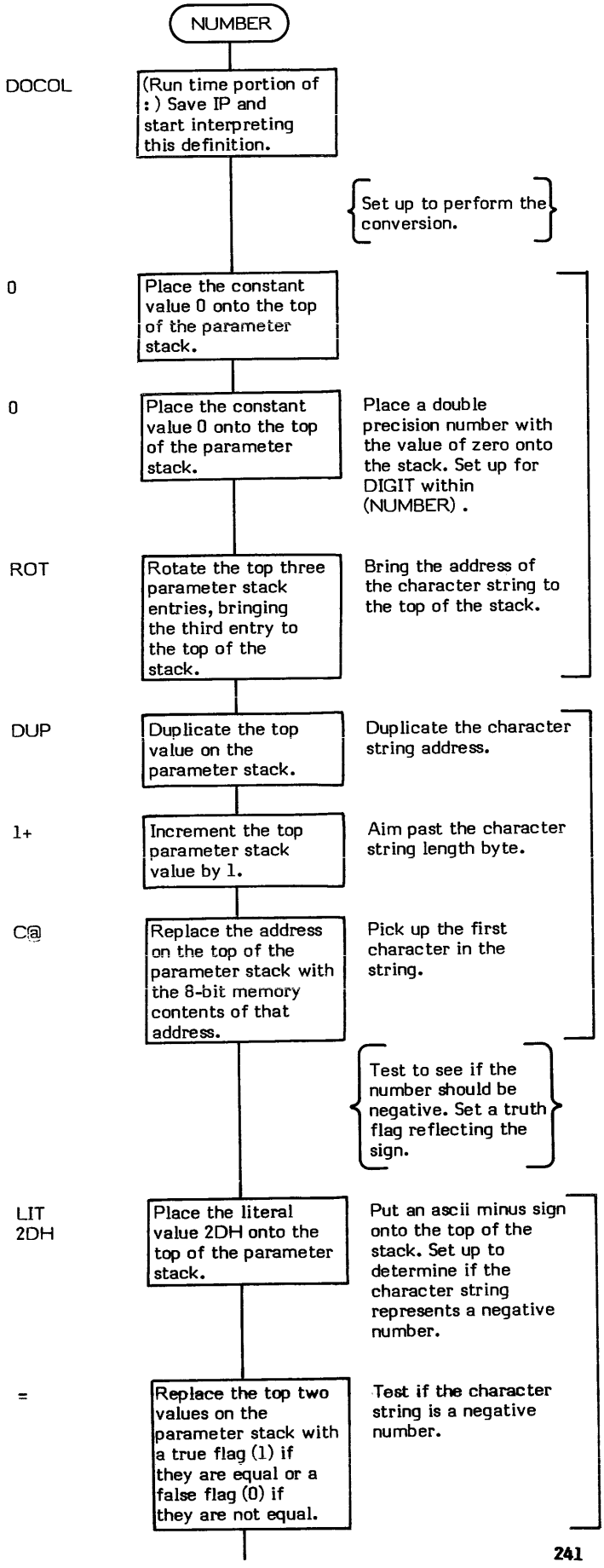
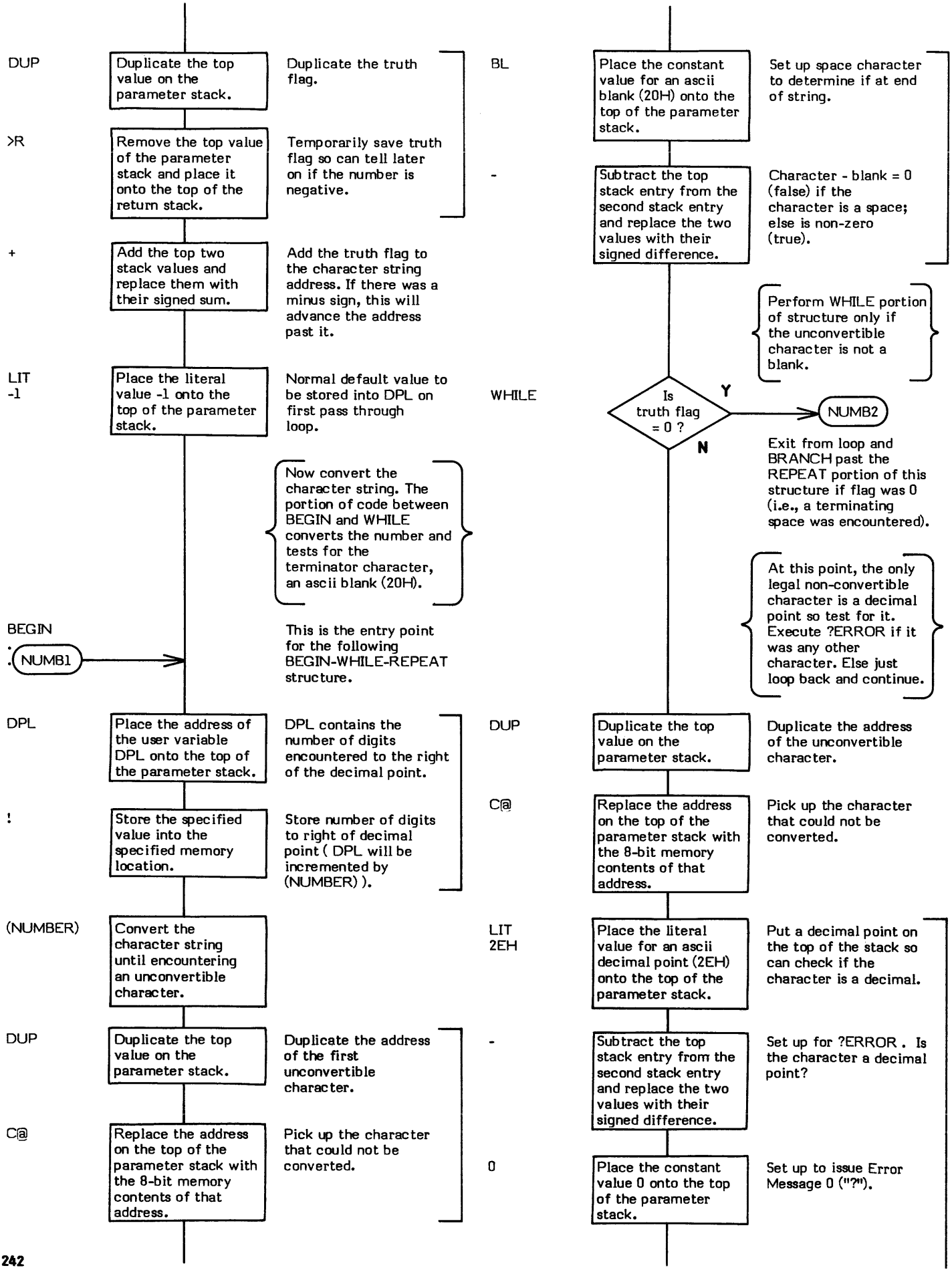


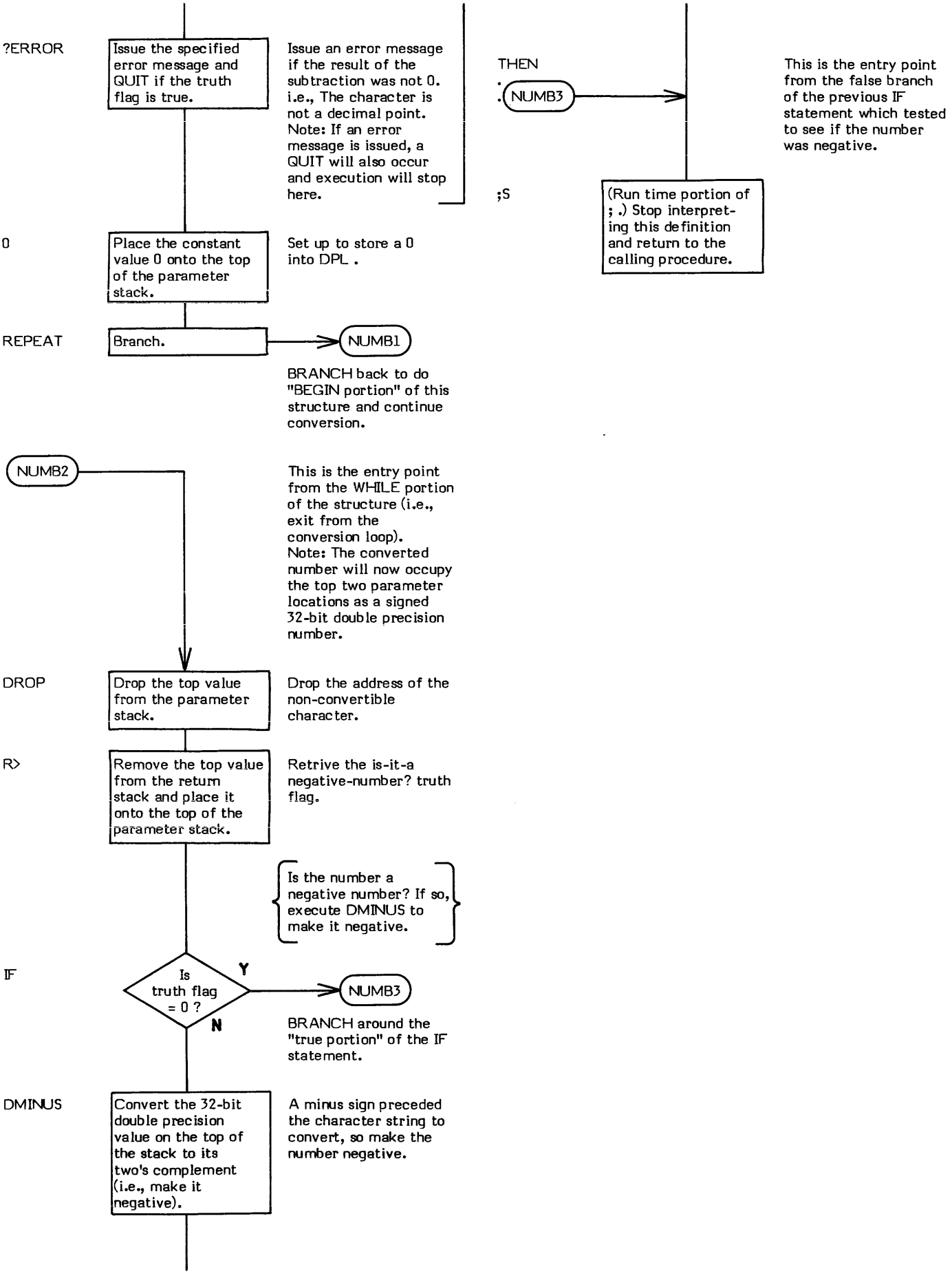
Figure NUMBER-1

High Level Logic of NUMBER

The logic boxes in this flowchart correspond roughly to the curly brackets in the low level flowchart.




DUP
Duplicate the top value on the parameter stack.
Duplicate the truth flag.
>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Temporarily save truth flag so can tell later on if the number is negative.
+
Add the top two stack values and replace them with their signed sum.
Add the truth flag to the character string address. If there was a minus sign, this will advance the address past it.
LIT
-1
Place the literal value -1 onto the top of the parameter stack.
Normal default value to be stored into DPL on first pass through loop.
Now convert the character string. The portion of code between BEGIN and WHILE converts the number and tests for the terminator character, an ascii blank (20H).
BEGIN
NUMB1
This is the entry point for the following BEGIN-WHILE-REPEAT structure.
DPL
Place the address of the user variable DPL onto the top of the parameter stack.
DPL contains the number of digits encountered to the right of the decimal point.
!
Store the specified value into the specified memory location.
Store number of digits to right of decimal point (DPL will be incremented by (NUMBER)).
(NUMBER)
Convert the character string until encountering an unconvertible character.
DUP
Duplicate the top value on the parameter stack.
Duplicate the address of the first unconvertible character.
C@
Replace the address on the top of the parameter stack with the 8-bit memory contents of that address.
Pick up the character that could not be converted.
BL
Place the constant value for an ascii blank (20H) onto the top of the parameter stack.
Set up space character to determine if at end of string.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Character - blank = 0 (false) if the character is a space; else is non-zero (true).
Perform WHILE portion of structure only if the unconvertible character is not a blank.
WHILE
Is truth flag = 0 ?
Y
N
NUMB2
Exit from loop and BRANCH past the REPEAT portion of this structure if flag was 0 (i.e., a terminating space was encountered).
At this point, the only legal non-convertible character is a decimal point so test for it. Execute ?ERROR if it was any other character. Else just loop back and continue.
DUP
Duplicate the top value on the parameter stack.
Duplicate the address of the unconvertible character.
C@
Replace the address on the top of the parameter stack with the 8-bit memory contents of that address.
Pick up the character that could not be converted.
LIT
2EH
Place the literal value for an ascii decimal point (2EH) onto the top of the parameter stack.
Put a decimal point on the top of the stack so can check if the character is a decimal.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
Set up for ?ERROR . Is the character a decimal point?
0
Place the constant value 0 onto the top of the parameter stack.
Set up to issue Error Message 0 ("?").


?ERROR
Issue the specified error message and QUIT if the truth flag is true.
Issue an error message if the result of the subtraction was not 0. i.e., The character is not a decimal point. Note: If an error message is issued, a QUIT will also occur and execution will stop here.
0
Place the constant value 0 onto the top of the parameter stack.
Set up to store a 0 into DPL .
REPEAT
Branch.
NUMB1
BRANCH back to do "BEGIN portion" of this structure and continue conversion.
NUMB2
This is the entry point from the WHILE portion of the structure (i.e., exit from the conversion loop). Note: The converted number will now occupy the top two parameter locations as a signed 32-bit double precision number.
DROP
Drop the top value from the parameter stack.
Drop the address of the non-convertible character.
R>
Remove the top value from the return stack and place it onto the top of the parameter stack.
Retrive the is-it-a negative-number? truth flag.
Is the number a negative number? If so, execute DMINUS to make it negative.
IF
Is truth flag = 0 ?
Y
N
NUMB3
BRANCH around the "true portion" of the IF statement.
DMINUS
Convert the 32-bit double precision value on the top of the stack to its two's complement (i.e., make it negative).
A minus sign preceded the character string to convert, so make the number negative.
THEN
.
.
NUMB3
This is the entry point from the false branch of the previous IF statement which tested to see if the number was negative.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

OFFSET

OFFSET (— data address)

OFFSET is a user variable that contains a block "offset" to mass-storage devices. Upon entry BLOCK adds the contents of OFFSET to the desired block number. This allows explicit selection of a specific device. i.e., The beginning block number of that device is stored into OFFSET. Future block references are then "offset" by that value and automatically reference the selected device.

DR0 and DR1 ("Drive 0" and "Drive 1") are device selection words that store beginning block numbers into OFFSET .

MESSAGE adjusts the block number of the message line so that the message is always relative to physical Drive 0 irregardless to the contents of OFFSET .

OFFSET is thoroughly covered in the description of BLOCK .

The user variable OFFSET is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable OFFSET .

Refer to BLOCK , DR0 , DR1 , MESSAGE , and USER .

FORTH-79: The FORTH-79 equivalent for OFFSET is OFFSET .

OR (value1 \ value2 — logical result)

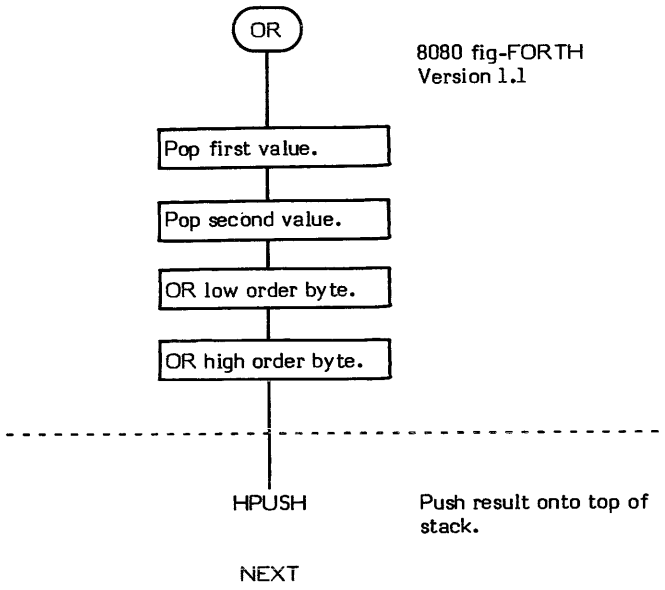
OR (pronounced "or") performs a bit-wise logical OR function on the top two values on the parameter stack and replaces them with their logical result.

UPDATE is an example of a word which uses OR .

* **At entry** - The first and second parameter stack entries both contain an absolute 16-bit single precision value to be ORed.
* **At exit** - The top of the parameter stack contains the absolute 16-bit single precision logical result.

OR is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for OR is OR .



OUT

OUT (— data address)

OUT is a user variable which contains a value incremented by EMIT . VLIST is the only system word that alters and references OUT . Applications may utilize OUT for purposes of formatting lines of text but note that EMIT only increments the value. No initialization or checking of OUT is performed by the sytem.

The user variable OUT is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable OUT .

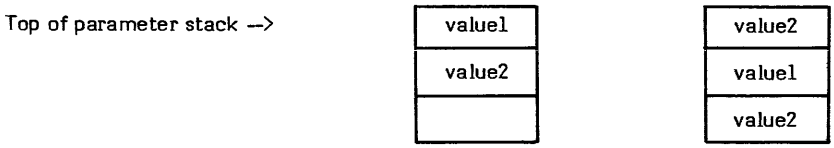
Refer to EMIT , VLIST , and USER .

FORTH-79: There is no FORTH-79 equivalent for OUT .

OVER (value2 \ value1 — value2 \ value1 \ value2)

OVER copies the second parameter stack entry onto the top of the parameter stack.

BEFORE AFTER

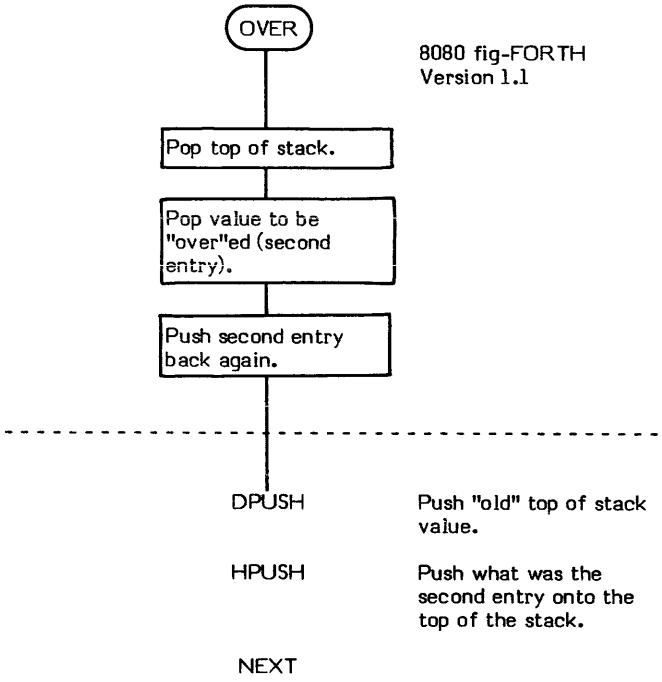


OVER is a very commonly used word. The definition WORD is an example of a word that uses OVER .

* **At entry** - The second parameter stack entry contains the 16-bit value to be copied onto the top of the stack.
* **At exit** - The value at the top of the parameter stack equals the value now at the third stack entry.

OVER is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for OVER is OVER .



PAD

PAD (— address)

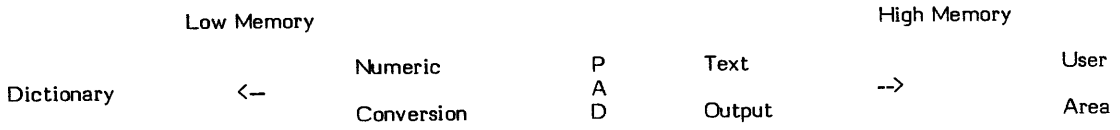
PAD (named for scratch PAD) places the address of the text output buffer onto the top of the parameter stack. PAD does not occupy a fixed location in memory; instead it is always a fixed offset away from the current end of the dictionary.

The PAD buffer is not a fixed length. It occupies memory between the end of the dictionary and the beginning of the user area. How much buffer is available at any time is a factor of both the installation memory map and the current dictionary size.

PAD is referenced two ways:

1. The pictured-numeric output words create numeric output text starting one byte before PAD and working towards low memory. (Refer to <# , # , #S , and #> .)

2. Application words such as text-editor words use PAD as a text buffer; with the beginning character stored in the first byte and the rest of the characters going towards high memory.



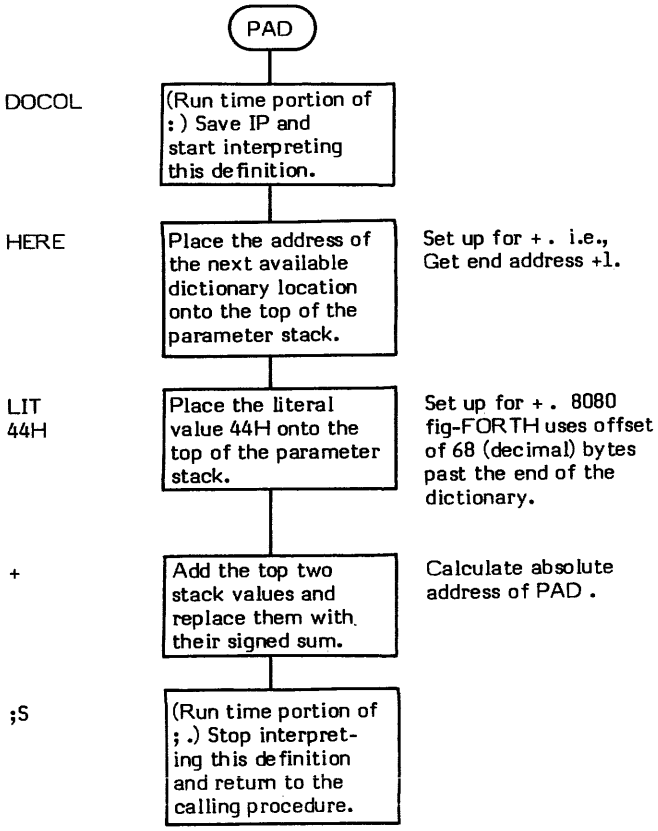
* **At entry** - No parameters.

* **At exit** - The top of the parameter stack contains the 16-bit beginning address of the PAD area.

PAD is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for PAD is PAD .

Definition:
```
: PAD ( -- address )
    HERE 44 +  ;
```



PFA (Name Field Address — Parameter Field Address)

PFA (pronounced "P-F-A") converts a given Name Field Address (NFA) of a dictionary definition into its Parameter Field Address (PFA).

The structure of the header of a FORTH definition is:

Name Field	Variable length
Link Field	2 byte address pointer
Code Field	2 byte address pointer
Parameter Field	Variable length

An example of the use of PFA can be found in the word (;CODE) .

* **At entry** - The top of the parameter stack contains the 16-bit Name Field Address of a FORTH definition.
* **At exit** - The top of the parameter stack contains the 16-bit Parameter Field Address of the specified FORTH definition.

NOTE: Since the fig-FORTH Name Field is a variable length, a simple subtraction cannot be used to directly calculate the Parameter Field Address. TRAVERSE must be used instead.

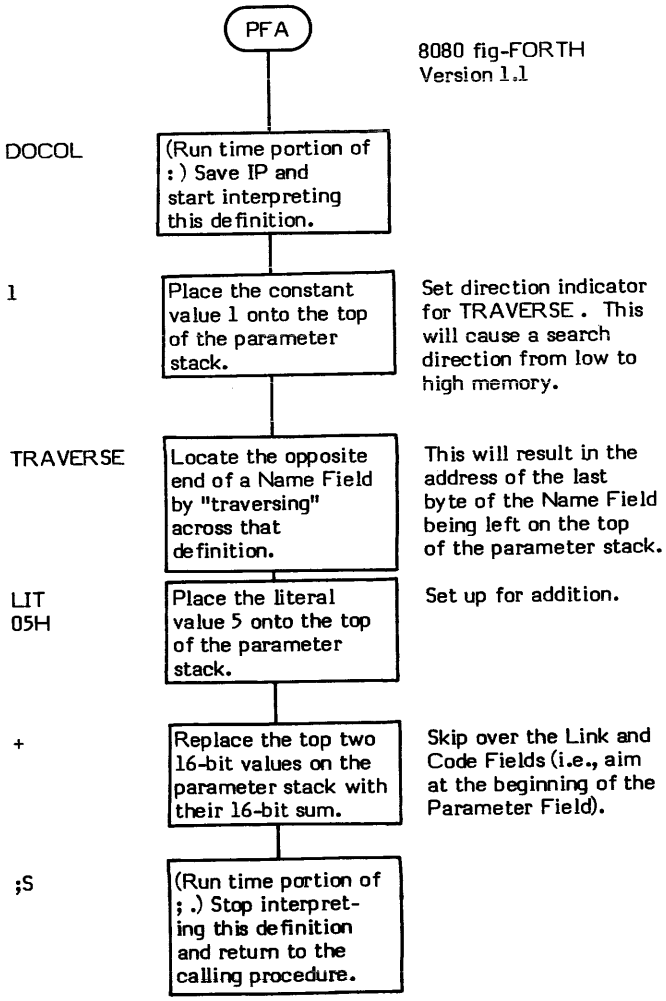
The exact nature of PFA may vary from system to system due to differing word sizes.

PFA is a high level colon definition.

Refer to NFA , CFA , and LFA .

FORTH-79: There is no FORTH-79 equivalent for PFA . A FORTH-79 program may not address into a definition's Parameter Field.

Definition:
```
:  PFA  ( NFA -- PFA )
     1 TRAVERSE 5 +  ;
```



PREV

PREV (— data address)

PREV (pronounced "PREV") is a system variable that contains the address of the most recently (PREViously) referenced disk buffer in the buffer-array. PREV is used by buffer-referencing management routines. The use of PREV is explained in detail in BLOCK .

Note that PREV is a system variable and not a user variable.

PREV is normally initialized to point to FIRST (the "first" buffer). In the fig-Model, this is done by using the "initial value" feature of the word VARIABLE (e.g., BUF1 VARIABLE PREV). This is not good programming practice and in the 8080 fig-FORTH Version 1.1 PREV is initialized by COLD .

The system variable PREV is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the system variable PREV .

Refer to BLOCK , BUFFER , +BUF , COLD , and VARIABLE .

FORTH-79: There is no FORTH-79 equivalent for PREV .

QUERY (--)

QUERY inputs up to 80 (decimal) characters of terminal text. Text input is prematurely halted by encountering a carriage return.

Text is placed into the Terminal Input Buffer, as specified by the contents of the user variable TIB . After text is input, the character pointer IN is set to 0. The carriage return is not stored in memory. A null (0) and a blank (20H) is appended to the end of the character string.

An example of the use of QUERY is in QUIT where QUERY is used to read in a line of text for INTERPRET .

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

EXECUTION ONLY (12H) -- This word must not be used while the system is in compile mode.

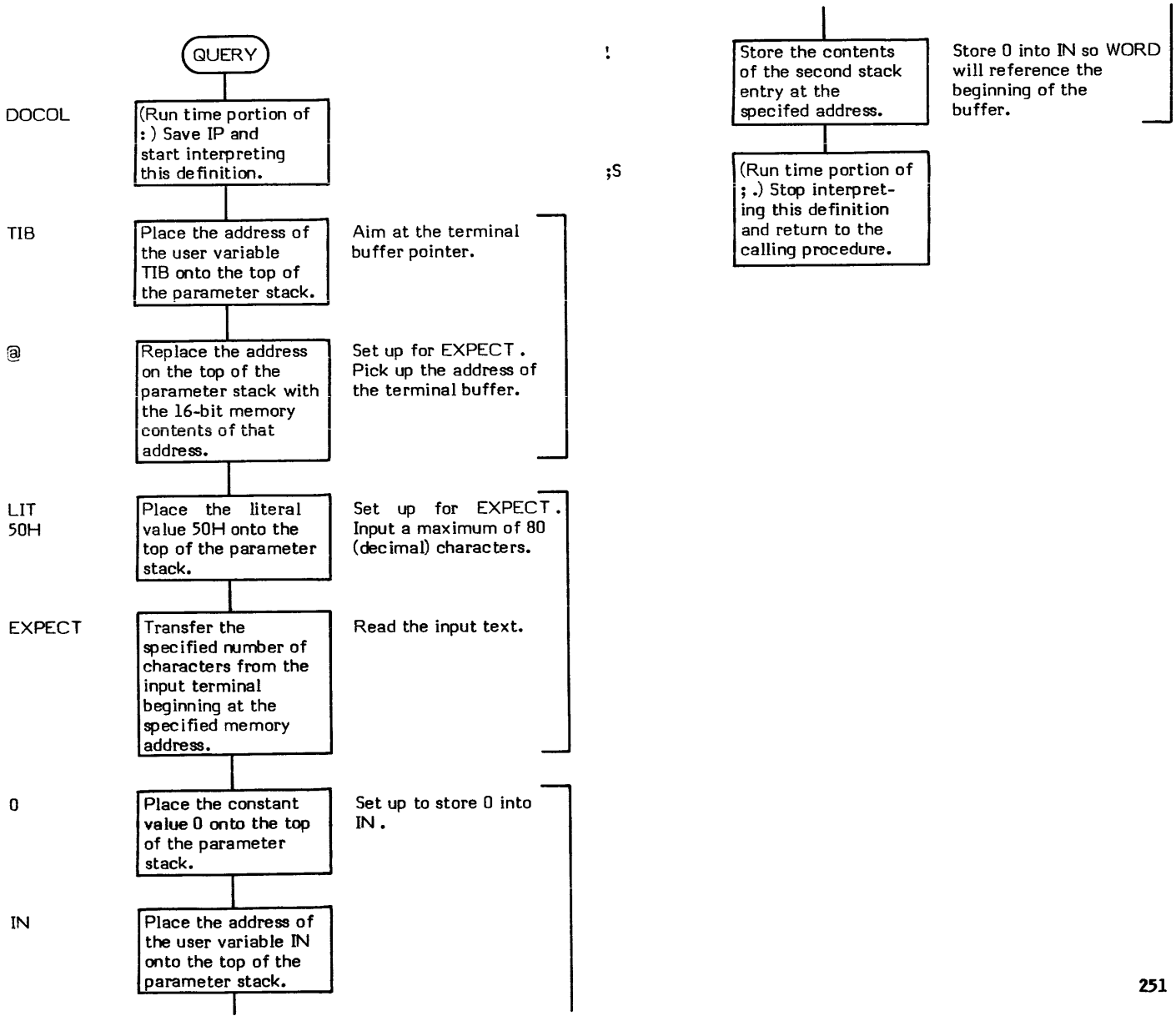
QUERY is a high level colon definition.

Refer to EXPECT , WORD , and TIB .

FORTH-79: The FORTH-79 equivalent for QUERY is QUERY .

Definition:

```
: QUERY ( -- )
    TIB @  50 EXPECT  0 IN !
```



QUIT

QUIT (–)

QUIT stops compilation, resets the return stack pointer, and starts interpretation from the input terminal data stream. The name "QUIT" is a rather ambiguous description of the function this procedure performs. Actually QUIT is used to initially start up FORTH, to "quit" execution of a routine, and to be the most basic and primary loop of the FORTH system. QUIT is an endless loop which calls the Interpreter repeatedly.

QUIT is an endless loop that inputs a line of text from the terminal and then calls the interpreter to process that text. After QUIT initially calls the Interpreter, commands from the terminal (residing in the Terminal Input Buffer) are executed which can cause any level of nesting. The eventual un-nesting of these levels always returns to QUIT . It is also possible to exit from any level of nesting by issuing QUIT (hence the name QUIT). This resets the return stack (i.e., absolutely erases any levels of nesting), sets input to come from the terminal, and starts INTERPRET .

When control returns from INTERPRET (see NULL), QUIT issues the prompt "OK" to the terminal.

* **At entry** - No parameters.
* **At exit** - No parameters.

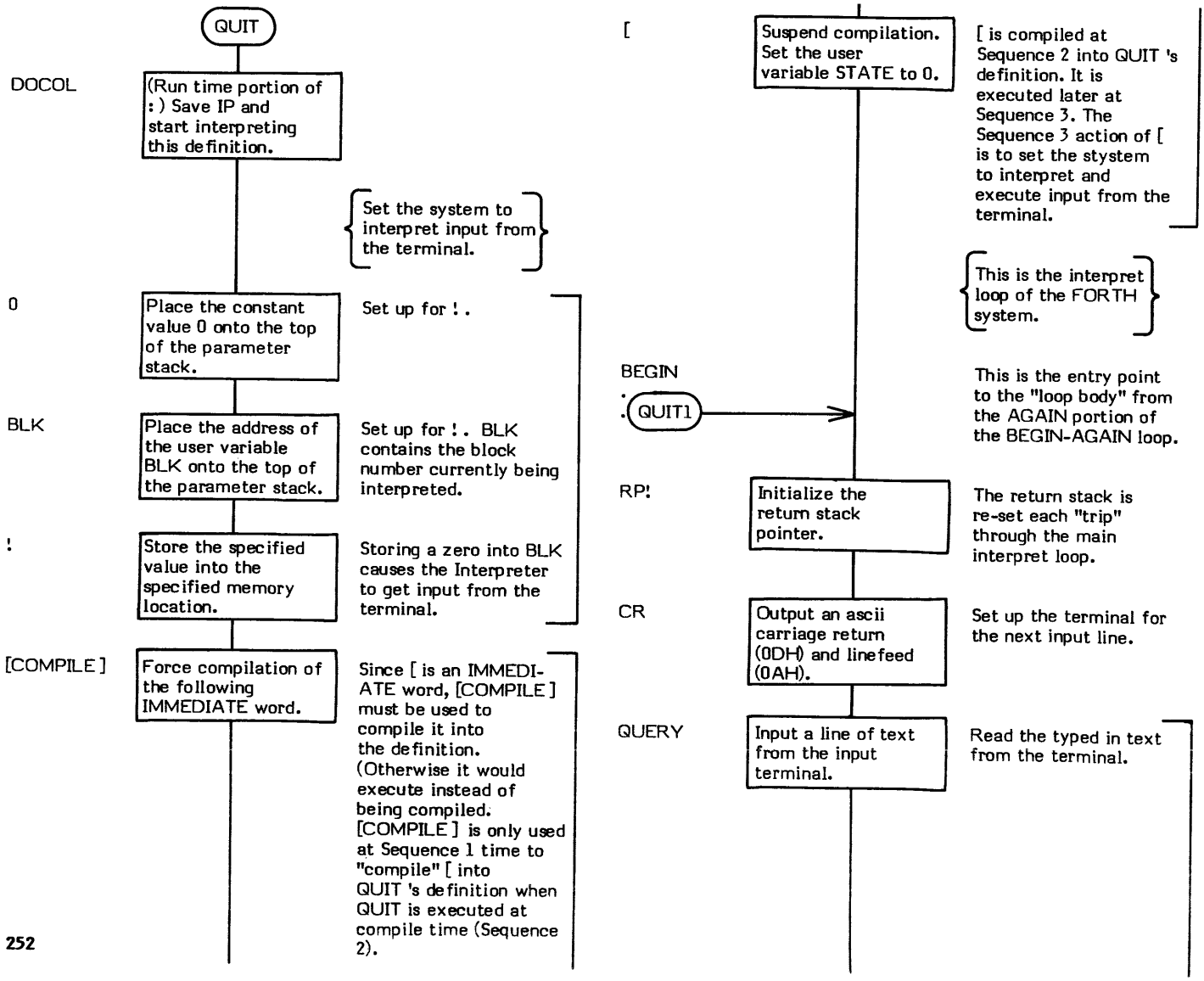
QUIT is a high level colon definition.

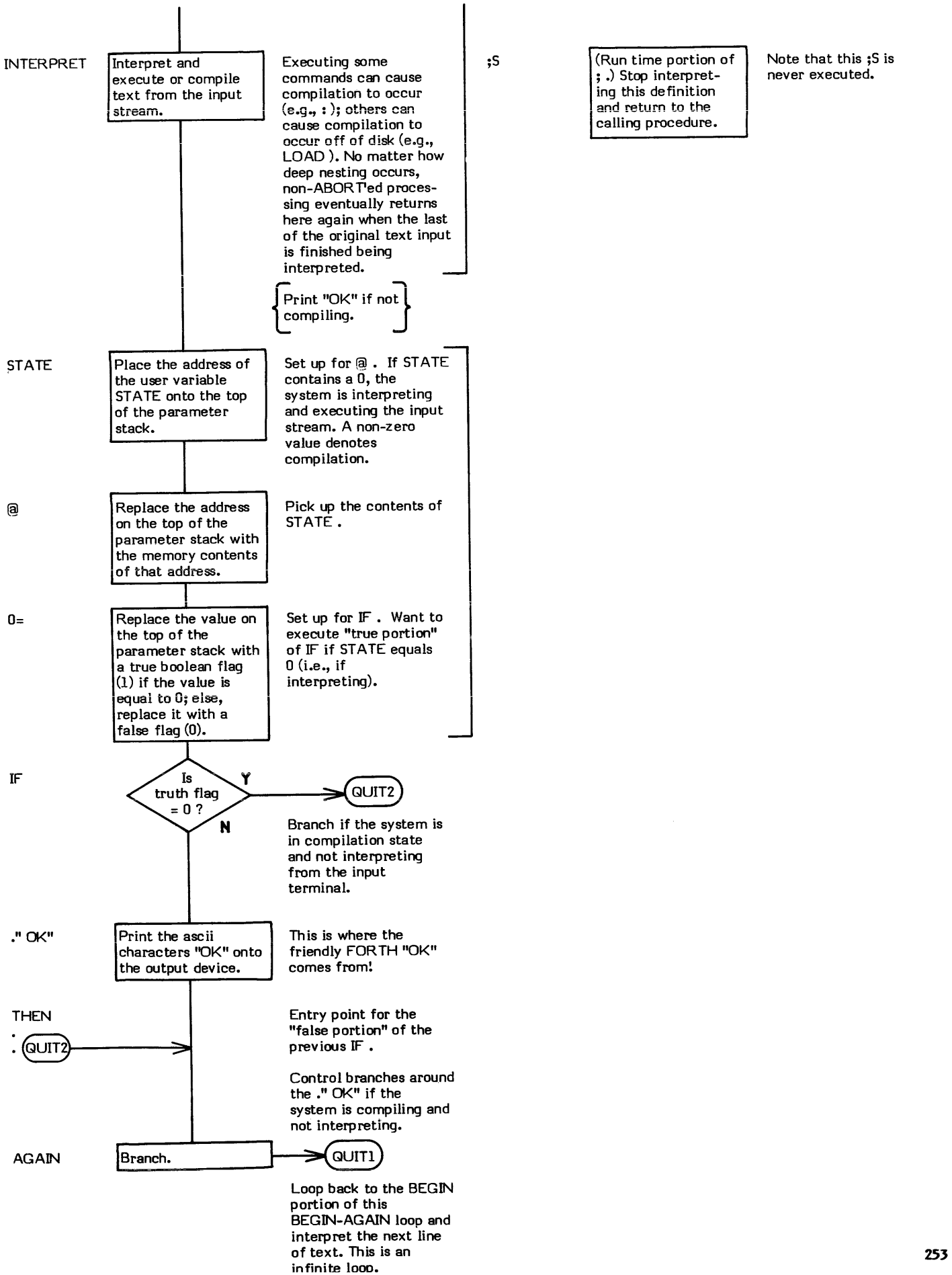
Refer to INTERPRET , and NULL .

FORTH-79: The FORTH-79 equivalent for QUIT is QUIT , although the FORTH-79 version does not output any message.

Definition:

```
: QUIT ( – )
    0 BLK !  [COMPILE] [
    BEGIN
        RP! CR  QUERY INTERPRET  STATE @ 0=
        IF  ." OK"  THEN
    AGAIN  ;
```




INTERPRET
Interpret and execute or compile text from the input stream.
Executing some commands can cause compilation to occur (e.g., :); others can cause compilation to occur off of disk (e.g., LOAD). No matter how deep nesting occurs, non-ABORT'ed processing eventually returns here again when the last of the original text input is finished being interpreted.
Print "OK" if not compiling.
STATE
Place the address of the user variable STATE onto the top of the parameter stack.
Set up for @ . If STATE contains a 0, the system is interpreting and executing the input stream. A non-zero value denotes compilation.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the contents of STATE .
0=
Replace the value on the top of the parameter stack with a true boolean flag (1) if the value is equal to 0; else, replace it with a false flag (0).
Set up for IF . Want to execute "true portion" of IF if STATE equals 0 (i.e., if interpreting).
IF
Is truth flag = 0 ?
Y
N
QUIT2
Branch if the system is in compilation state and not interpreting from the input terminal.
." OK"
Print the ascii characters "OK" onto the output device.
This is where the friendly FORTH "OK" comes from!
THEN
:
.
QUIT2
Entry point for the "false portion" of the previous IF .
Control branches around the ." OK" if the system is compiling and not interpreting.
AGAIN
Branch.
QUIT1
Loop back to the BEGIN portion of this BEGIN-AGAIN loop and interpret the next line of text. This is an infinite loop.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
Note that this ;S is never executed.

R

R (— **value on top of return stack**)

R copies the top of the return stack to the top of the parameter stack. In this sense, it performs the same function as I (although, due to installation differences, I should still be used with "do-loops" to perform this function).

Note that R does not "drop" the top of the return stack and therefore cannot be used to compensate for the use of >R .

(.") is an example of a word which uses R.

* **At entry** - No parameter stack entries but the top of the return stack contains the 16-bit value to be copied.
* **At exit** - The top of the parameter stack contains the copied 16-bit value.

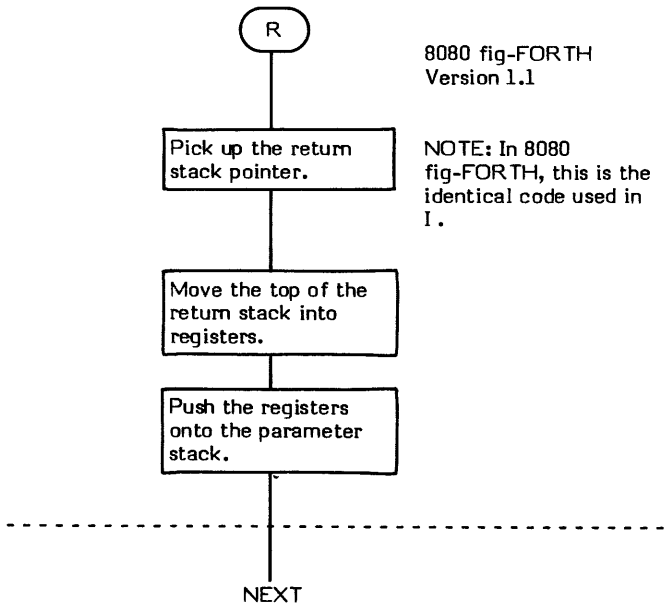
R is a low level code primitive.

Refer to I .

FORTH-79: The FORTH-79 equivalent for R is R@ .

Definition:

```
:   R/W    ( buffer address  block number  R/W flag -- )    ( 8080 Version 1.1 )
      USE >R    SWAP  SEC/BLK *    ROT  USE !
      SEC/BLK 0          DO
                         OVER  OVER  T&SCAL  IF
                                                  SECRD
                                             ELSE
                                                  SECWR
                                             THEN
                         1+    80  USE  +!
                     LOOP
    DROP  DROP    R> USE !    ;
```



8080 fig-FORTH
Version 1.1

NOTE: In 8080 fig-FORTH, this is the identical code used in I .

R# **(— data address)**

R# (pronounced "R-sharp") is a user variable that may be used by file related functions such as editing. R# is not used in the basic FORTH system.

The user variable R# is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit address of the user variable R# .

Refer to USER .

FORTH-79: There is no FORTH-79 equivalent for R# .

R/W

R/W (buffer address \ desired block number \ R/W flag —)

R/W (pronounced "R-slash-W") is the standard fig-FORTH mass-storage interface word. All mass storage data transfers are performed by this word. In general, I/O words below BLOCK are system dependent. The following description is for the 8080 fig-FORTH Version 1.1 under a CP/M environment. This is, however, a very good example of an implementation of R/W .

The word has two major functions:

1. To convert the specified virtual block number into an absolute address on a physical device.

2. To perform the actual data transfer between the device and memory.

The conversion of a block number (which to the programmer represents data residing in memory) into a physical device address is the heart of the virtual aspect of FORTH I/O. Any type, or combinations of types of mass storage devices may be used; although for practical purposes each device must be randomly accessible. The most common mass storage devices are floppy disk, hard disk, and magnetic tape.

Device selection is based solely upon block number range. Each storage device in the system has an unique range of block numbers assigned to it. This allows each block in the system to be unique. For example, Disk Drive 1 may contain blocks 0 through 799; while Disk Drive 2 may contain blocks 800 through 1199; and Tape Drive 3 may contain blocks 1200 through 1599.

The word BLOCK provides a mechanism where drives may be implicitly referenced by biasing the desired block number with the starting block number of the selected device. (See BLOCK , OFFSET , DR0 , and DR1 .) Note that by the time R/W is called, this block number must have been converted to an absolute block number.

The conversion by R/W of this absolute block number into a physical hardware address is obviously completely dependent on the characteristics of the selected device.

Likewise, the actual I/O driver routines will vary from system to system. R/W may contain any number of I/O drivers for whatever devices are connected to the system.

Note that, since the I/O drivers are installation dependent, any I/O error checking routines are also installation dependent. FORTH does not have to perform unverified I/O. Any desired degree of error checking or retries may be put into R/W .

* **At entry** - The top of the parameter stack contains a Read or Write boolean flag. A zero flag indicates that a write is to occur; a non-zero flag, that a read is to occur. The second stack entry contains an absolute 16-bit absolute block number. This value may range from 0 to 32,767. The third stack entry contains the 16-bit buffer address of the beginning of the data to be transferred.

* **At exit** - No parameters.

R/W is a high level colon definition.

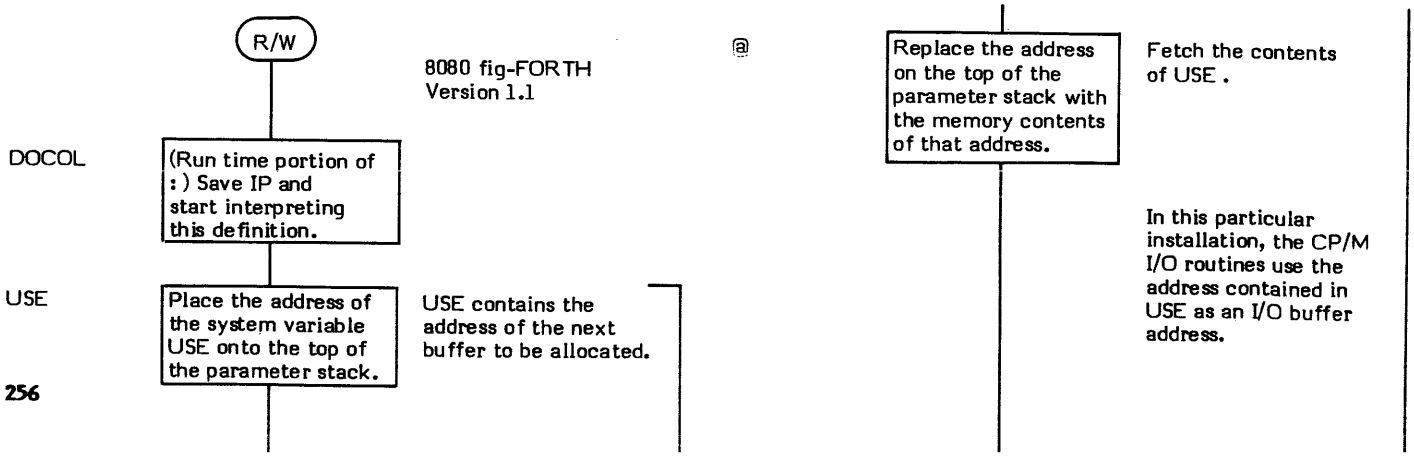
Refer to BLOCK , +BUF , BUFFER , OFFSET , DR0 , and DR1 .

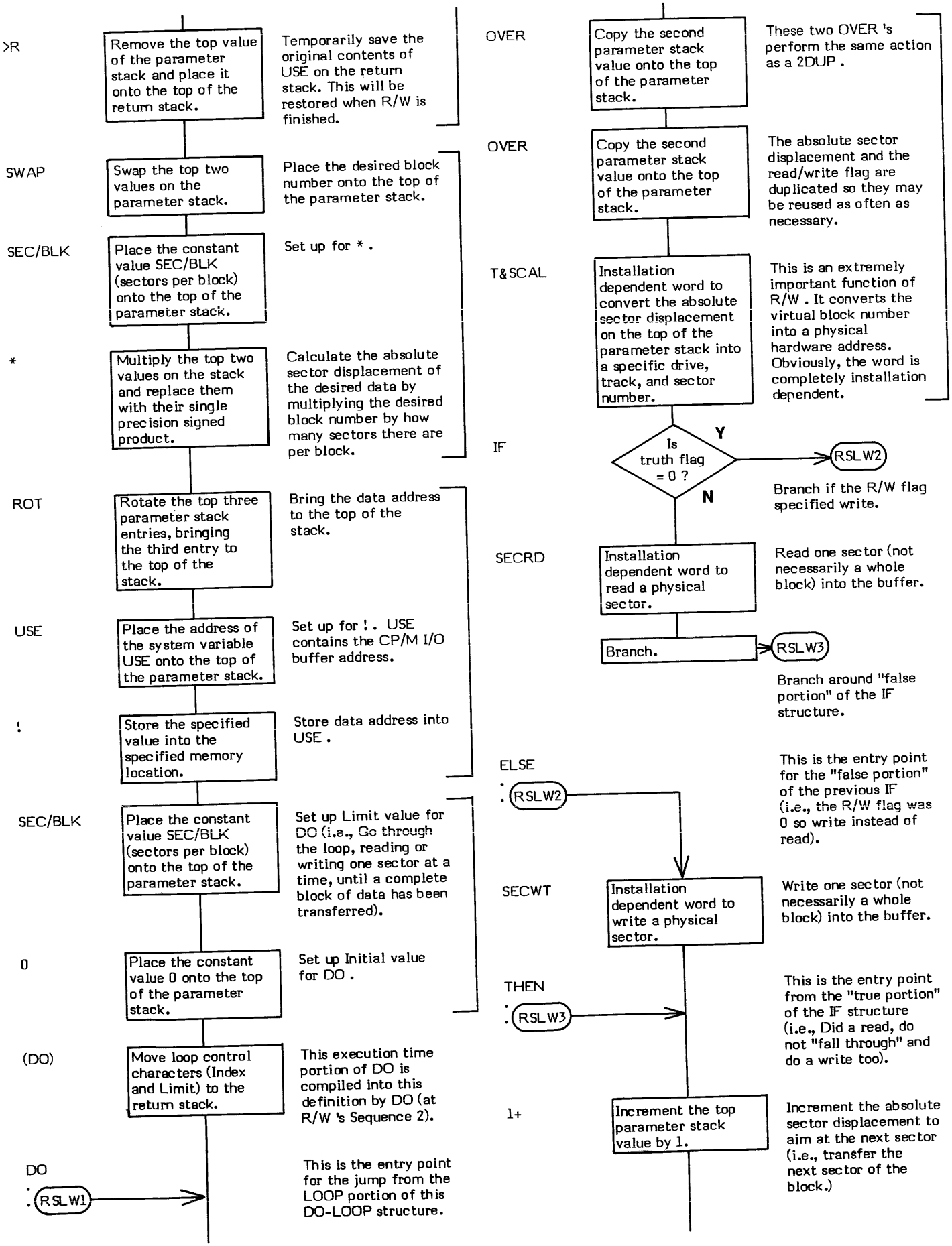
FORTH-79: There is no FORTH-79 equivalent for R/W .

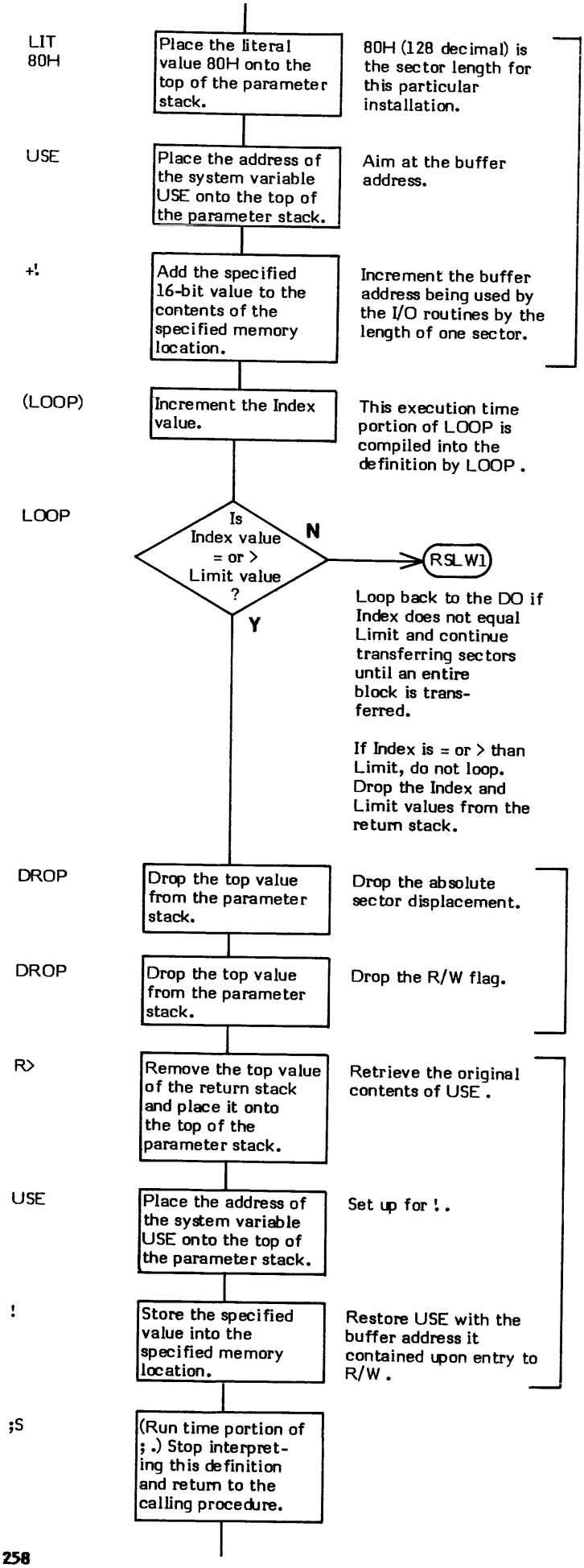
Definition:

```
:  R/W   ( buffer address  block number  R/W flag — )    ( 8080 Version 1.1 )
      USE >R    SWAP  SEC/BLK *    ROT  USE !
      SEC/BLK  0        DO
                        OVER  OVER  T&SCAL  IF
                                              SECRD
                                        ELSE
                                              SECWR
                                        THEN
                              1+    80  USE +!
                        LOOP
   DROP  DROP    R>  USE !    ;
```

This is an example of a R/W implementation for the 8080 fig-FORTH Version 1.1 in a CP/M environment.




>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Temporarily save the original contents of USE on the return stack. This will be restored when R/W is finished.
SWAP
Swap the top two values on the parameter stack.
Place the desired block number onto the top of the parameter stack.
SEC/BLK
Place the constant value SEC/BLK (sectors per block) onto the top of the parameter stack.
Set up for * .
*
Multiply the top two values on the stack and replace them with their single precision signed product.
Calculate the absolute sector displacement of the desired data by multiplying the desired block number by how many sectors there are per block.
ROT
Rotate the top three parameter stack entries, bringing the third entry to the top of the stack.
Bring the data address to the top of the stack.
USE
Place the address of the system variable USE onto the top of the parameter stack.
Set up for ! . USE contains the CP/M I/O buffer address.
!
Store the specified value into the specified memory location.
Store data address into USE .
SEC/BLK
Place the constant value SEC/BLK (sectors per block) onto the top of the parameter stack.
Set up Limit value for DO (i.e., Go through the loop, reading or writing one sector at a time, until a complete block of data has been transferred).
0
Place the constant value 0 onto the top of the parameter stack.
Set up Initial value for DO .
(DO)
Move loop control characters (Index and Limit) to the return stack.
This execution time portion of DO is compiled into this definition by DO (at R/W 's Sequence 2).
DO
RSLW1
This is the entry point for the jump from the LOOP portion of this DO-LOOP structure.
OVER
Copy the second parameter stack value onto the top of the parameter stack.
These two OVER 's perform the same action as a 2DUP .
OVER
Copy the second parameter stack value onto the top of the parameter stack.
The absolute sector displacement and the read/write flag are duplicated so they may be reused as often as necessary.
T&SCAL
Installation dependent word to convert the absolute sector displacement on the top of the parameter stack into a specific drive, track, and sector number.
This is an extremely important function of R/W . It converts the virtual block number into a physical hardware address. Obviously, the word is completely installation dependent.
IF
Is truth flag = 0 ?
Y
N
RSLW2
Branch if the R/W flag specified write.
SECRD
Installation dependent word to read a physical sector.
Read one sector (not necessarily a whole block) into the buffer.
Branch.
RSLW3
Branch around "false portion" of the IF structure.
ELSE
RSLW2
This is the entry point for the "false portion" of the previous IF (i.e., the R/W flag was 0 so write instead of read).
SECWT
Installation dependent word to write a physical sector.
Write one sector (not necessarily a whole block) into the buffer.
THEN
RSLW3
This is the entry point from the "true portion" of the IF structure (i.e., Did a read, do not "fall through" and do a write too).
1+
Increment the top parameter stack value by 1.
Increment the absolute sector displacement to aim at the next sector (i.e., transfer the next sector of the block.)


LIT
80H
Place the literal value 80H onto the top of the parameter stack.
80H (128 decimal) is the sector length for this particular installation.
USE
Place the address of the system variable USE onto the top of the parameter stack.
Aim at the buffer address.
+!
Add the specified 16-bit value to the contents of the specified memory location.
Increment the buffer address being used by the I/O routines by the length of one sector.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
RSLW1
Y
Loop back to the DO if Index does not equal Limit and continue transferring sectors until an entire block is transferred.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
DROP
Drop the top value from the parameter stack.
Drop the absolute sector displacement.
DROP
Drop the top value from the parameter stack.
Drop the R/W flag.
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Retrieve the original contents of USE .
USE
Place the address of the system variable USE onto the top of the parameter stack.
Set up for ! .
!
Store the specified value into the specified memory location.
Restore USE with the buffer address it contained upon entry to R/W .
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

R0 (-- data address)

R0 (pronounced "R-zero") is a user variable which contains the initial address of the return stack.

R0 is initialized by COLD during system start up with data from the origin parameter area.

QUIT uses the address in R0 to reset the return stack.

The user variable R0 is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit address of the user variable R0 .

Refer to QUIT , COLD , and USER .

FORTH-79: There is no FORTH-79 equivalent for R0 .

R>

R> (— value popped from return stack)

R> (pronounced "R-from") pops a number off of the return stack and places it onto the top of the parameter stack. It is often used in conjunction with >R in order to restore the return stack to its original state.

NOTE: The incorrect use of R> can cause indeterminate results. Most likely the system would crash. Although not "illegal," the use of R> outside of a colon definition will probably cause a system crash.

R> differs from R in that the value is removed, or "popped", from the top of the return stack rather than copied.

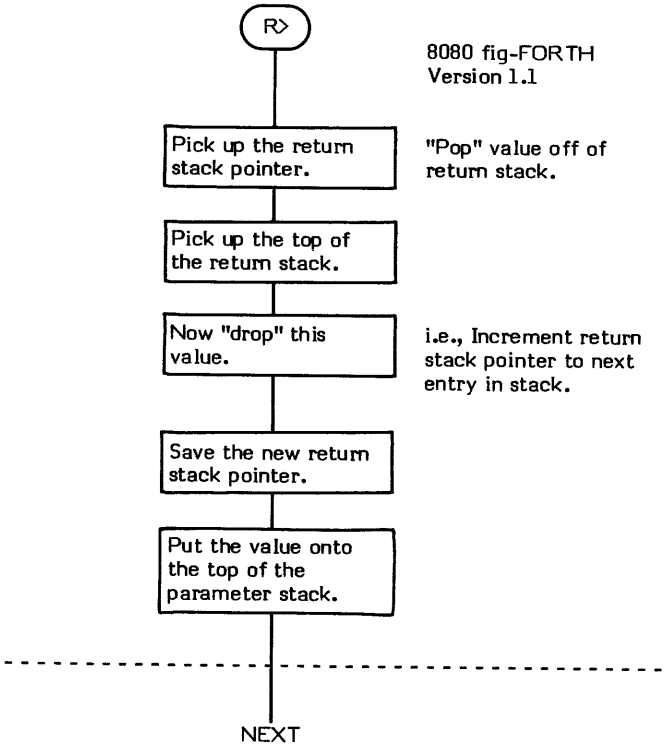
REPEAT is an example of a word which uses R>.

* **At entry** - No parameter stack entries. The top of the return stack contains the 16-bit value to be transferred.
* **At exit** - The top of the parameter stack contains the 16-bit value which was previously on the top of the return stack.

R> is a low level code primitive.

Refer to >R , and R .

FORTH-79: The FORTH-79 equivalent for R> is R> .



REPEAT

COMPILE TIME (Sequence 2): (loop address \ 1 \ offset location \ 4 --)

EXECUTION TIME (Sequence 3): (--)

REPEAT is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

REPEAT is used to mark the end of a BEGIN-WHILE-REPEAT structure in the form:

```
BEGIN
"Set Exit Conditional"
WHILE
"True Portion of Loop Body"
REPEAT
```

Refer to WHILE for a description of the action of a BEGIN-WHILE-REPEAT loop.

At compile time (Sequence 2), REPEAT primarily does two things:

1. REPEAT compiles the unconditional branch back to BEGIN (i.e., the loop action of the structure).

2. REPEAT resolves and stores the exit branch offset located in WHILE. Upon exit, WHILE uses this branch to exit to the word immediately following REPEAT .

The unconditional branch is compiled by executing an AGAIN , which inputs the loop address and the value 1 left by the BEGIN . The processing necessary to resolve the exit offset in WHILE is identical to that of resolving the 0BRANCH in an IF ; therefore, an ENDIF is used. ENDIF expects the value 2 on the stack for compiler security. REPEAT subtracts 2 from the value on the stack and since WHILE left the value 4, it is likely that an unmatched WHILE and REPEAT will be detected.

The run time action (Sequence 3) of REPEAT is to cause an unconditional BRANCH back to the first word after BEGIN . Then whatever processing necessary to set the exit boolean flag can be performed and the loop repeated.

BEGIN-WHILE-REPEAT loop structures must be used within a colon definition.

Note that REPEAT is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

NUMBER is an example of a word which uses REPEAT.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 4. This value is left on the stack by WHILE and is tested by REPEAT for compiler security to ensure that the BEGIN-WHILE-REPEAT structure does contain a WHILE . The second stack entry contains the 16-bit address of the reserved 0BRANCH offset location created by the IF inside of WHILE . The third stack entry contains the 16-bit signed single precision value 1 left by the BEGIN to ensure that the structure is started with a BEGIN . The fourth stack entry contains the 16-bit address of the first location following the BEGIN statement (i.e., the loop address).

* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.

* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

REPEAT is a high level colon definition.

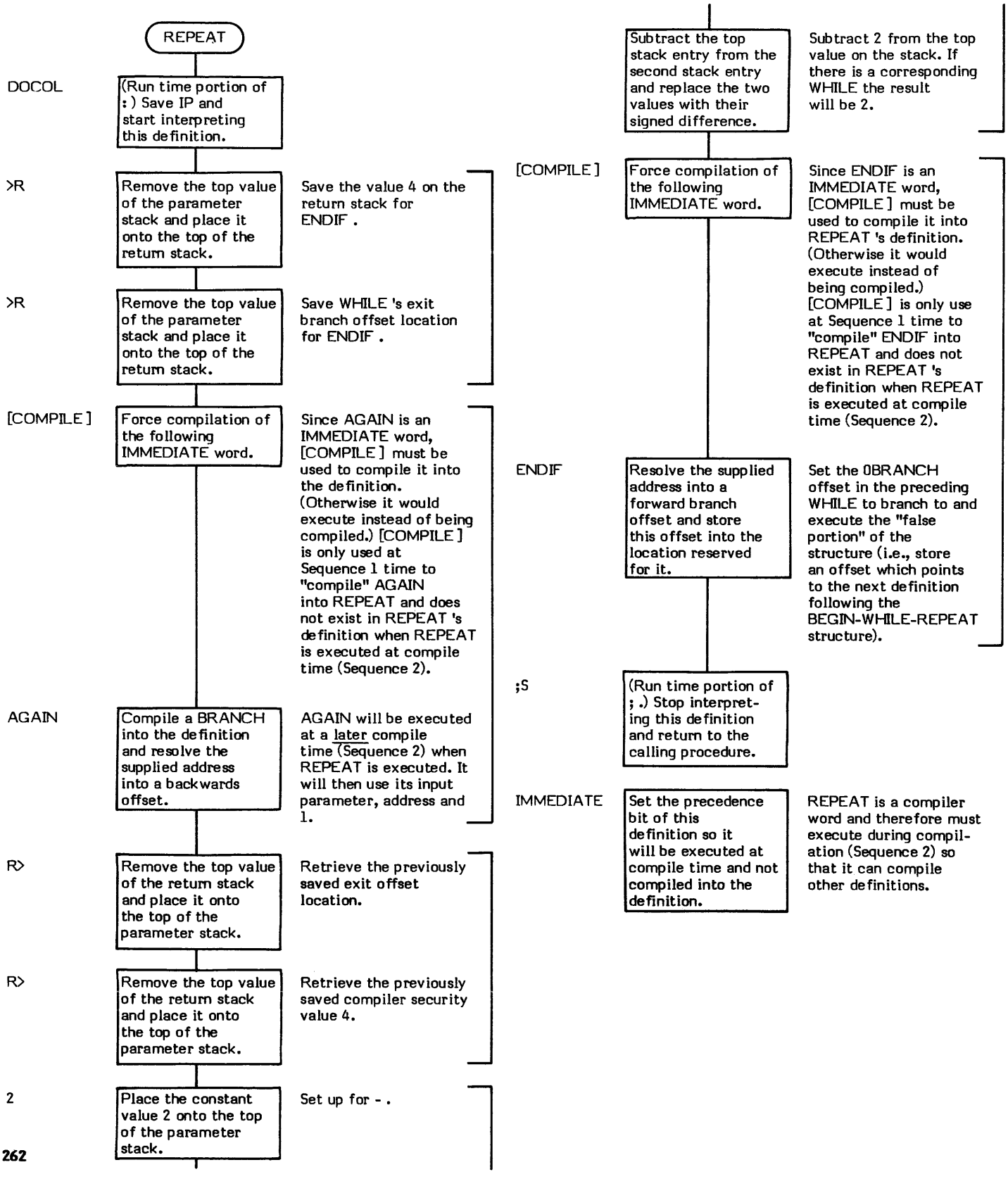
Refer to BEGIN , WHILE , AGAIN , ENDIF and BRANCH .

FORTH-79: The FORTH-79 equivalent for REPEAT is REPEAT .

Definition:

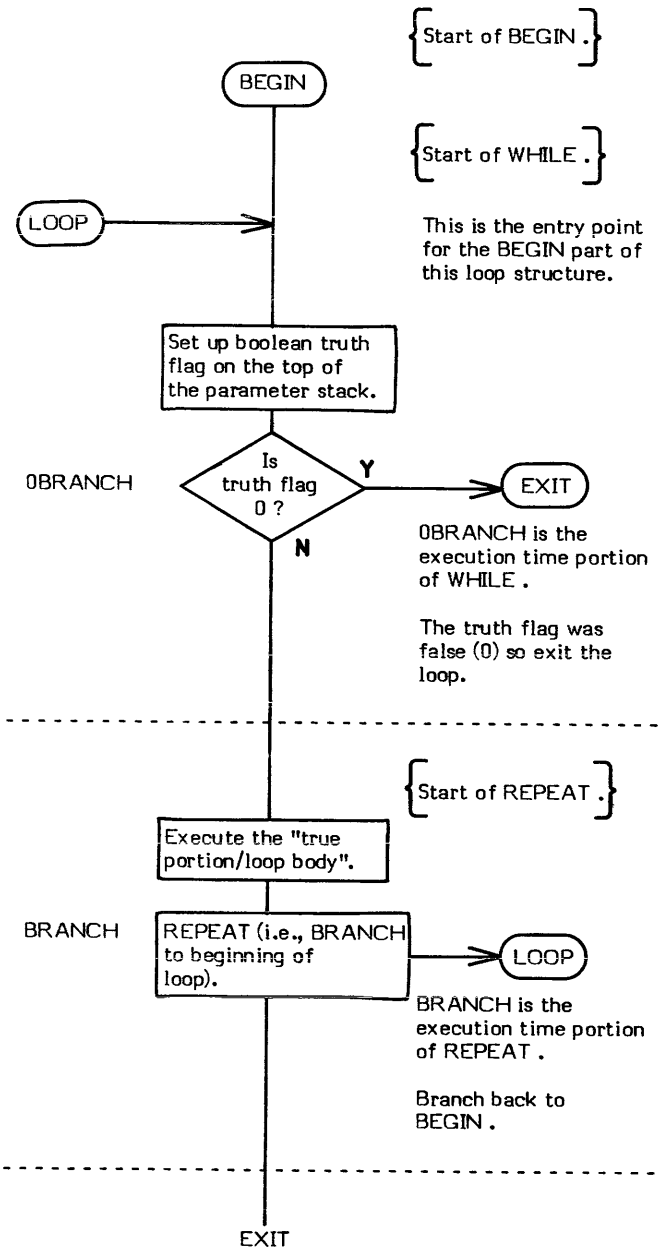
```
:  REPEAT  ( loop address \ 1 \ offset location \ 4 -- )
     >R >R  [COMPILE] AGAIN  R> R>  2 -  [COMPILE] ENDIF  ;  IMMEDIATE
```

COMPILE TIME action of REPEAT Sequence 2): (loop address \ 1 \ offset location \ 4 --)



EXECUTION TIME action of REPEAT (Sequence 3): (—)

Repeat is the "loop back" portion of a BEGIN-WHILE-REPEAT loop.



ROT

ROT (value1 \ value2 \ value 3 — value2 \ value3 \ value1)

ROT (pronounced "rote") rotates the top three 16-bit values on the parameter stack. The third value is placed onto the top of the stack.

	BEFORE	AFTER
Top of Parameter Stack ----->	value3	value1
	value2	value3
	value1	value2

NUMBER is an example of a word that uses ROT .

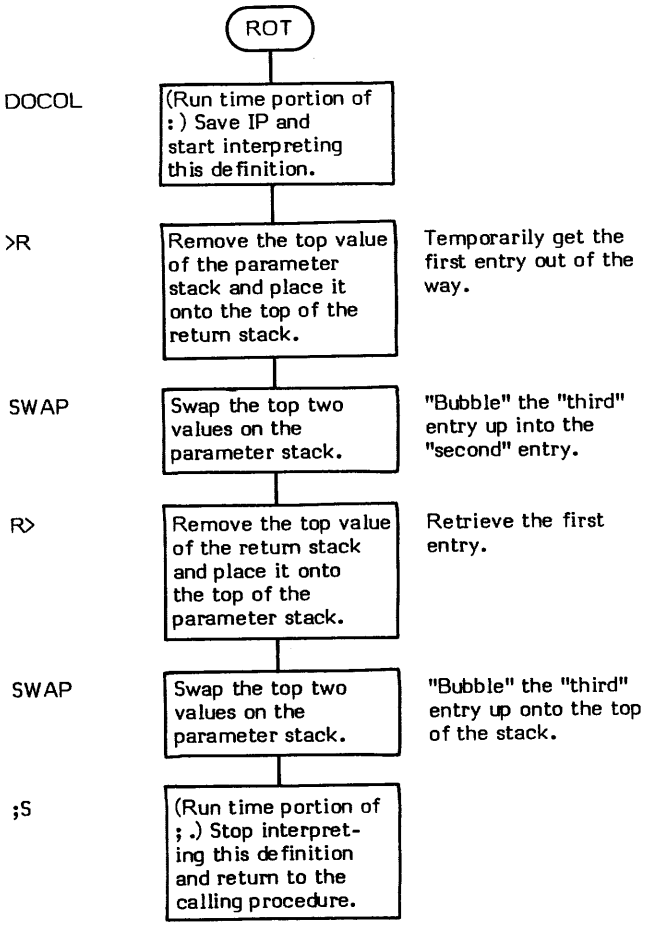
* **At entry** - The parameter stack contains three 16-bit values to be rotated.
* **At exit** -The 16-bit value which was previously the third stack entry is not on the top of the stack. The first and second stack entries are each one deeper in the stack.

ROT is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for ROT is ROT .

Definition:

```
:  ROT  ( value1 \ value2 \ value3 -- value2 \ value3 \ value1 )
     >R  SWAP  >R  SWAP  ;
```



RP! (–)

RP! (pronounced "R-P-store") is an installation dependent word that initializes the return stack pointer to the address contained in the user variable R0. This is the commonly used method of initializing the return stack pointer.

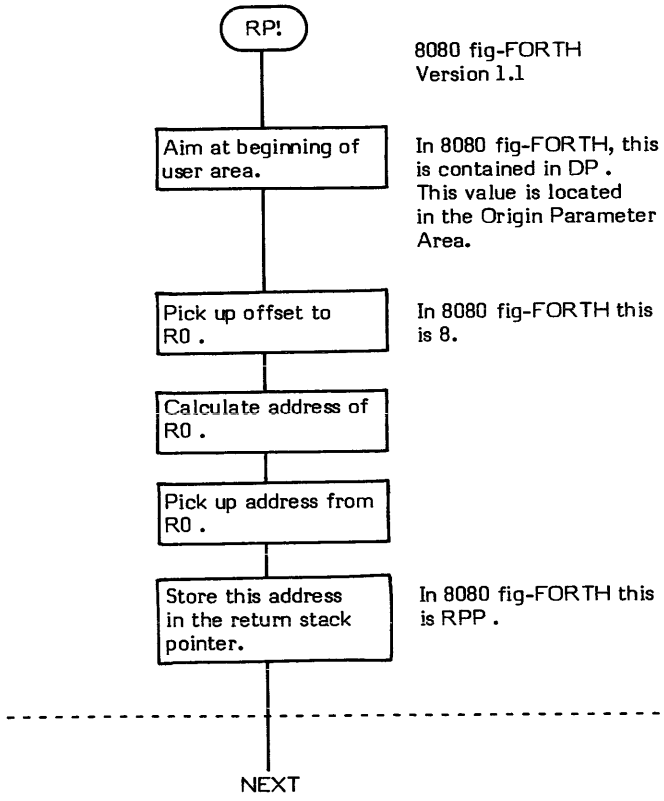
RP! is used in COLD and QUIT .

* **At entry** - No parameters.
* **At exit** - No parameters.

RP! is a low level code primitive.

Refer to R0 .

FORTH-79: There is no FORTH-79 equivalent for RP! .



RP@

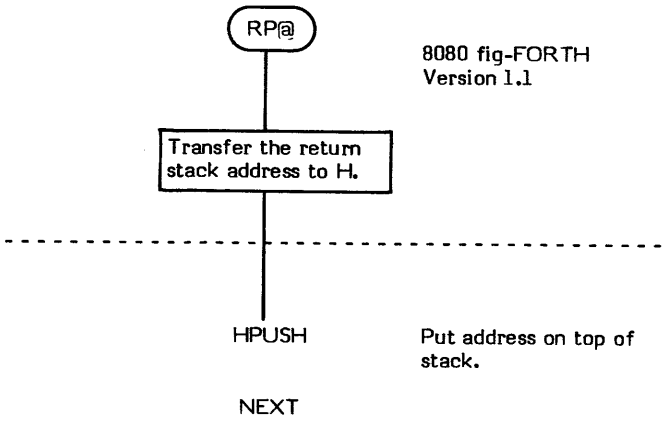
RP@ (— return stack pointer address)

RP@ (pronounced "R-P-fetch") is an installation dependent word that returns the return stack pointer address, present at the time RP@ was initially invoked, to the top of the parameter stack.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the 16-bit address of the top of the return stack.

RP@ is a low level code primitive.

FORTH-79: There is no FORTH-79 equivalent for RP@ .



S->D (single precison number — double precision number)

S->D (pronounced "S-to-D") extends a signed single precision number to form a signed double precision number.

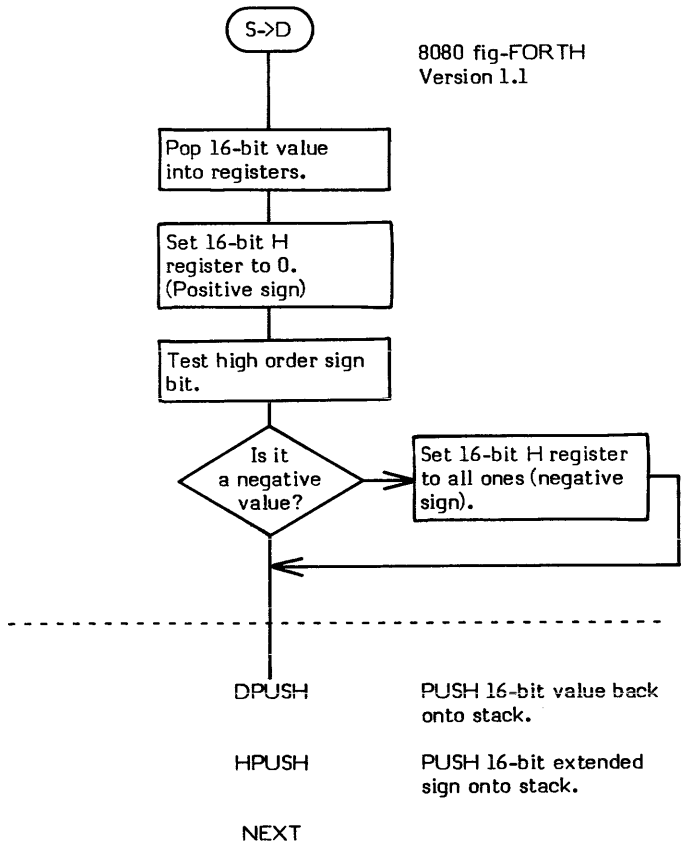
WARNING! S->D must be used to convert single precion values to double precision values in order to correctly propagate the sign bits.

/MOD is an example of a word that uses S->D .

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value.
* **At exit** - The top of the parameter stack contains a signed 32-bit double precision value with the high order word (containing the extended sign) on the top of the stack and the original value in the second stack entry.

S->D is a low level code primitive.

FORTH-79: There is no FORTH-79 equivalent for S->D .



S0

S0 (— data address)

S0 (pronounced "S-zero") is a user variable that contains the initial address of the parameter stack. S0 is initialized by COLD during system start up with data from the origin parameter area.

ABORT uses the address in S0 to reset the parameter stack.

SP! fetches the contents of S0 when it initializes the parameter stack.

The user variable S0 is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable S0 .

Refer to ABORT ,COLD , SP! and USER .

FORTH-79: The FORTH-79 equivalent for S0 is S0 .

SCR (— data address)

SCR (pronounced "S-C-R") is a user variable that contains the screen number most recently referenced by LIST . This value can then be referenced by other words--especially editor commands--so that the desired screen number does not have to be explicitly stated every time a command is issued.

The user variable SCR is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable SCR .

Refer to LIST , and USER .

FORTH-79: The FORTH-79 equivalent for SCR is SCR .

SIGN

SIGN (sign flag \ double precision value — double precision value)

SIGN is normally used within a pictured numeric output expression to place a sign immediately to the left of a converted numeric character string.

SIGN will place an ascii minus sign in the next available character string location if the sign of the third parameter stack entry (i.e., a "sign flag") is negative. The magnitude of the sign flag entry is ignored. The top two entries contain a double precision value.

The sign flag parameter is normally set up before the <# #S SIGN #> expression is executed. Generally the sign flag is the same sign as the double precision value to be converted. To set up for <# #S SIGN #> , the words SWAP OVER DABS will leave a copy of the high order signed portion of the value in the third stack entry. This is what is done in D.R which uses SIGN for signed pictured numeric output. The description of <# and # explains more about pictured numeric conversion.

* **At entry** - The third entry of the parameter stack must contain a signed 16-bit value which acts as a sign flag. The top two entries contain a 32-bit double precision value.

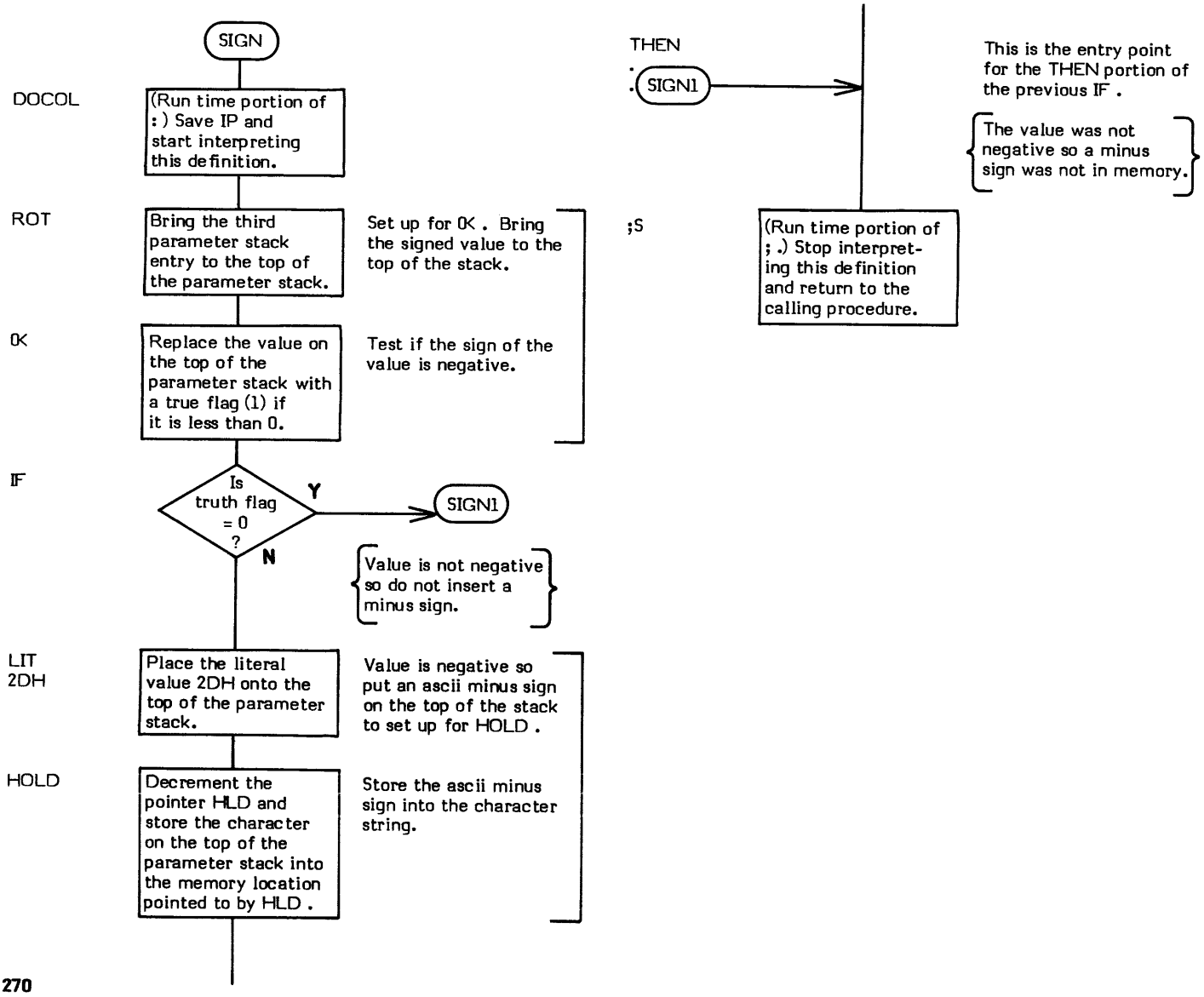
* **At exit** - The original third entry, the sign flag, in the parameter stack is dropped. The first and second stack entries remain untouched.

SIGN is a high level colon definition.

Refer to # , <# , #> , #S , HOLD , HLD , PAD , and DABS .

FORTH-79: The FORTH-79 equivalent for SIGN is SIGN .

Definition: : SIGN (flag \ double precision value -- double precision value)
ROT 0< IF 2D HOLD THEN



SMUDGE (–)

SMUDGE is used during word definition to toggle the "smudge" bit (binary mask value 20H) in the length byte of the Name Field of a definition so that the word cannot be "found" (via the word (FIND)) until the word is correctly compiled. The length byte is smudged at the beginning of compilation and is again smudged (toggled back) upon successful completion (usually by ;).

* **At entry** - No parameters.
* **At exit** - No parameters.

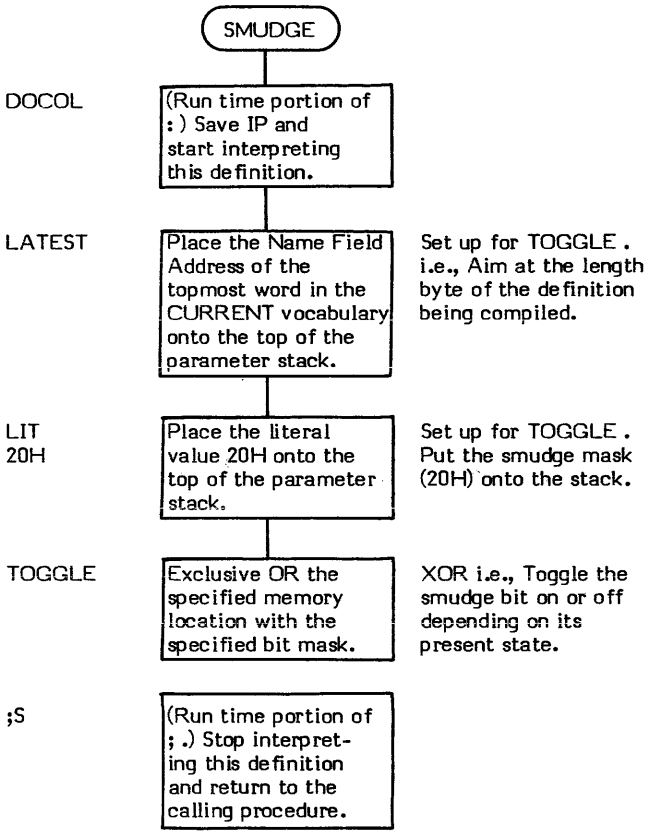
SMUDGE is a high level colon definition.

Refer to ; .

FORTH-79: There is no FORTH-79 equivalent for SMUDGE .

Definition:

```
:  SMUDGE  ( -- )
       LATEST  20  TOGGLE
```



SP!

SP! (–)

SP! (pronounced "S-P-store") is an installation dependent word that initializes the parameter stack pointer to the address contained in the user variable S0. This is the commonly used method of initializing the stack pointer.

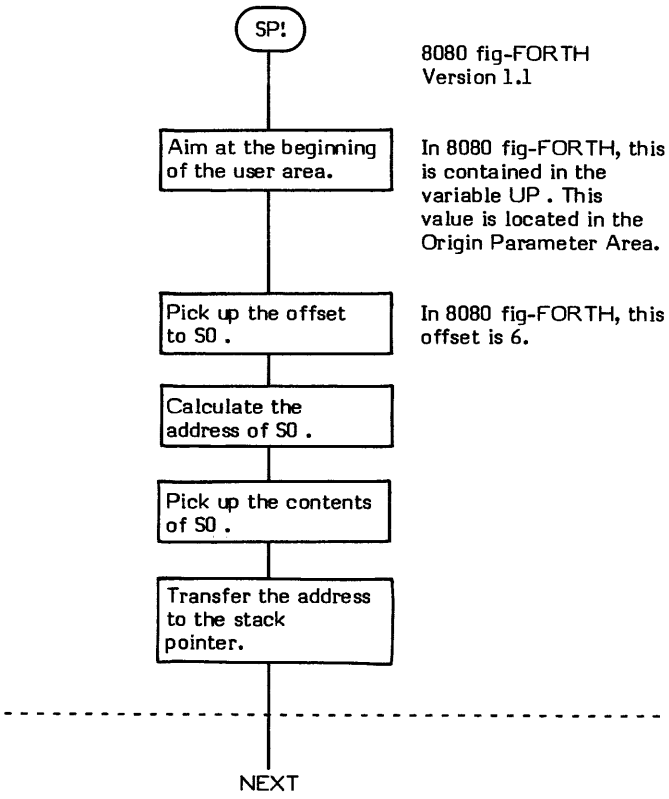
ABORT and ERROR are examples of words that use SP! .

* **At entry** - No parameters.
* **At exit** - No parameters.

SP! is a low level code primitive.

Refer to S0 , ABORT , and ERROR .

FORTH-79: There is no FORTH-79 equivalent for SP! .



SP@ (— parameter stack pointer address)

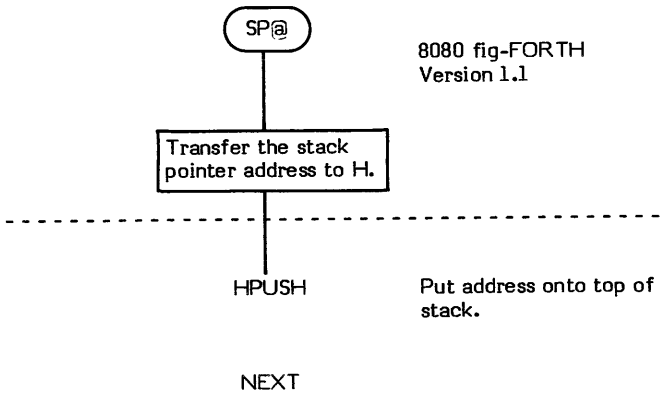
SP@ (pronounced "S-P-fetch") is an installation dependent word that returns the parameter stack pointer address, present at the time SP@ was initially invoked, to the top of the stack.

!CSP is a word that uses SP@ .

* **At entry** - No parameters.
* **At exit** - The top of the stack contains the 16-bit stack pointer address.

SP@ is a low level code primitive.

FORTH-79: SP@ is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".



SPACE

SPACE (–)

SPACE transmits an ascii blank (20H) to the output device.

D. is an example of a word that uses SPACE .

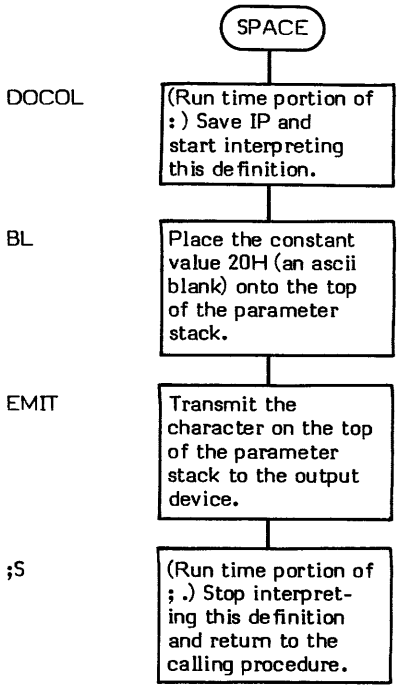
* **At entry** - No parameters.
* **At exit** -No parameters.

SPACE is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for SPACE is SPACE .

Definition:

```
: SPACE ( -- )
    BL EMIT ;
```



SPACES (count —)

SPACES outputs a specified number of ascii blanks (20H) to the output device.

D.R is an example of a word that uses SPACES .

* **At entry** - The top of the parameter stack contains a signed 16-bit single precision value indicating the number of blanks to output.

  NOTE: Negative values are changed to zero.

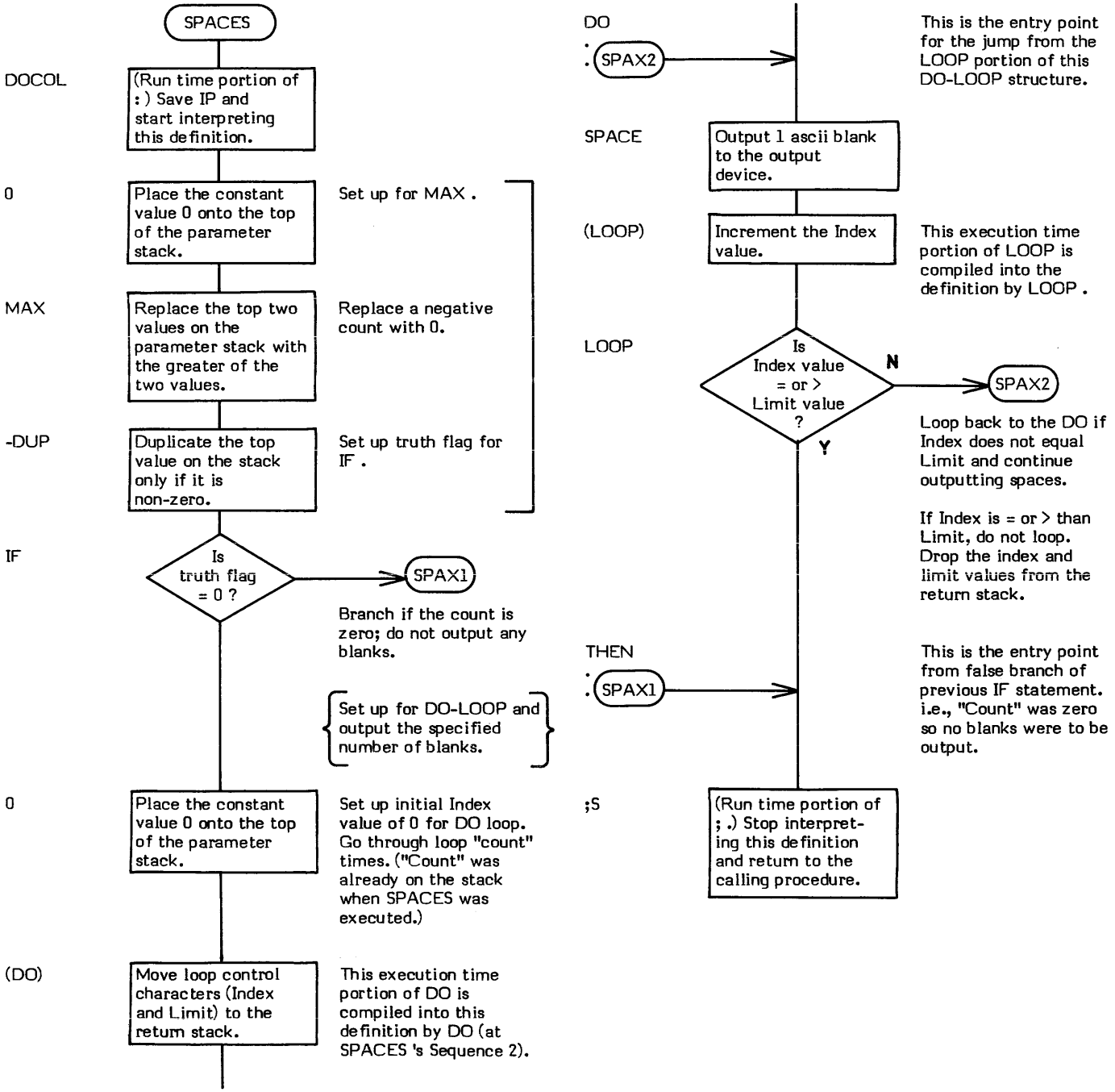
* **At exit** - No parameters.

SPACES is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for SPACES is SPACES .

Definition:

```
: SPACES ( count — )
    0 MAX -DUP IF 0 DO SPACE LOOP THEN ;
```



STATE

STATE (— data address)

STATE is a user variable which contains a value that reflects the compilation "state" of the system.

A 0 value indicates that the system is compiling. A non-zero value (C0H for the 8080 fig-FORTH Version 1.1) indicates that the system is interpreting.

INTERPRET refers to the value in STATE to determine how to process the input data stream.

[sets STATE to interpret mode.

] sets STATE to compile mode.

During system start up, STATE is originally initialized by QUIT .

The user variable STATE is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable STATE .

Refer to [,] , INTERPRET , QUIT , and USER .

FORTH-79: The FORTH-79 equivalent for STATE is STATE .

SWAP (**value2 \valuel — valuel \ value2**)

SWAP exchanges the top two 16 bit values on the parameter stack.

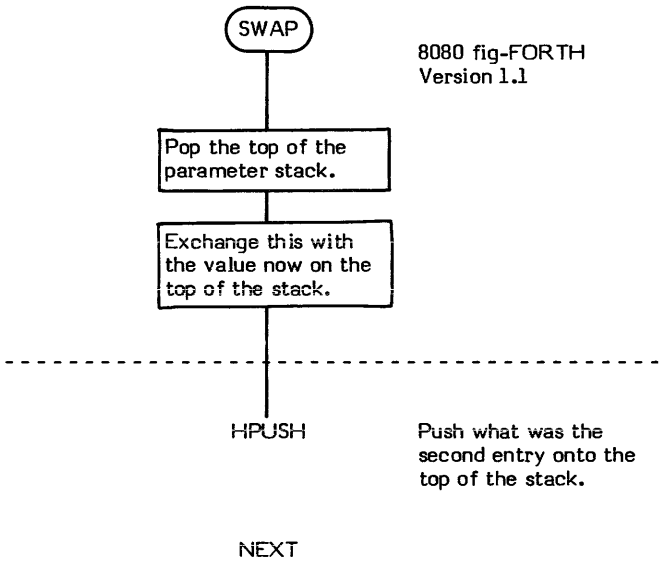
	Before	After
Top of Parameter Stack --->	valuel value2	value2 valuel

SWAP is a very commonly used word. ROT is an example of a word that uses SWAP .

* **At entry** - The top of the parameter stack and the second entry contain the 16-bit values to be exchanged.
* **At exit** -The top of the stack contains the value previously in the second entry. The second entry will contain the value previously on the top of the stack.

SWAP is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for SWAP is SWAP .



TASK

TASK (–)

TASK is a no-op (no operation) definition that is used as a "boundary marker" between applications or program segments. The word simply consists of a colon, a name, and a semicolon. For example:

```
: TASK ;
```

TASK is normally compiled as the first word of an application, with all other application words following it in the dictionary. A future FORGET TASK will then forget all of the application up to (and including) the boundary defined by TASK .

* **At entry** - No parameters.
* **At exit** - No parameters.

TASK is a high level colon definition.

Refer to FORGET .

FORTH-79: There is no FORTH-79 equivalent for TASK .

THEN

COMPILE TIME (Sequence 1): (offset address\ 2 --)

EXECUTION TIME (Sequence 3): (--)

THEN is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

THEN is used to make the end of an IF-THEN or IF-ELSE-THEN structure. (ENDIF may be used in place of THEN but THEN is the preferred word.) THEN must be used in the form:

IF "true portion" THEN

IF "true portion" ELSE "false portion" THEN

An IF-THEN structure must be used within a colon definition.

THEN is actually a Sequence 1 compiling word. That is THEN compiles (during Sequence 1) another compiling word, ENDIF , into its definition. Then, at Sequence 2, when THEN is executed the word ENDIF does the actual compiling into the definition being created.

The apparent Sequence 2 compile time action of THEN is to compute a forward branch offset, calculated from the supplied input parameter address to the next available memory location following THEN (supplied by HERE) and to store that offset in the location reserved by the previous IF or ELSE statement. (Actually this is performed at Sequence 2 via the ENDIF which was compiled by THEN at Sequence 1.)

Some compiler security is provided by checking for a 2 on the top of the stack. A THEN without a corresponding IF or ELSE will probably not encounter a 2 on the top of the stack during compilation. (NOTE: During compilation, IF and ELSE leave a 2 on the top of the stack.)

The execution time action (Sequence 3) of THEN is to simply serve as the destination of a forward branch from a previous IF or ELSE statement. THEN compiles no run time code.

Note that THEN is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 2 used for compiler security. The second stack entry contains a 16-bit address specifying both the branch offset address of a previous IF or ELSE and also the memory location that offset is to be stored into.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) -- There is some sort of problem with the pairing of conditionals within the definition being compiled.

THEN is a high level colon definition.

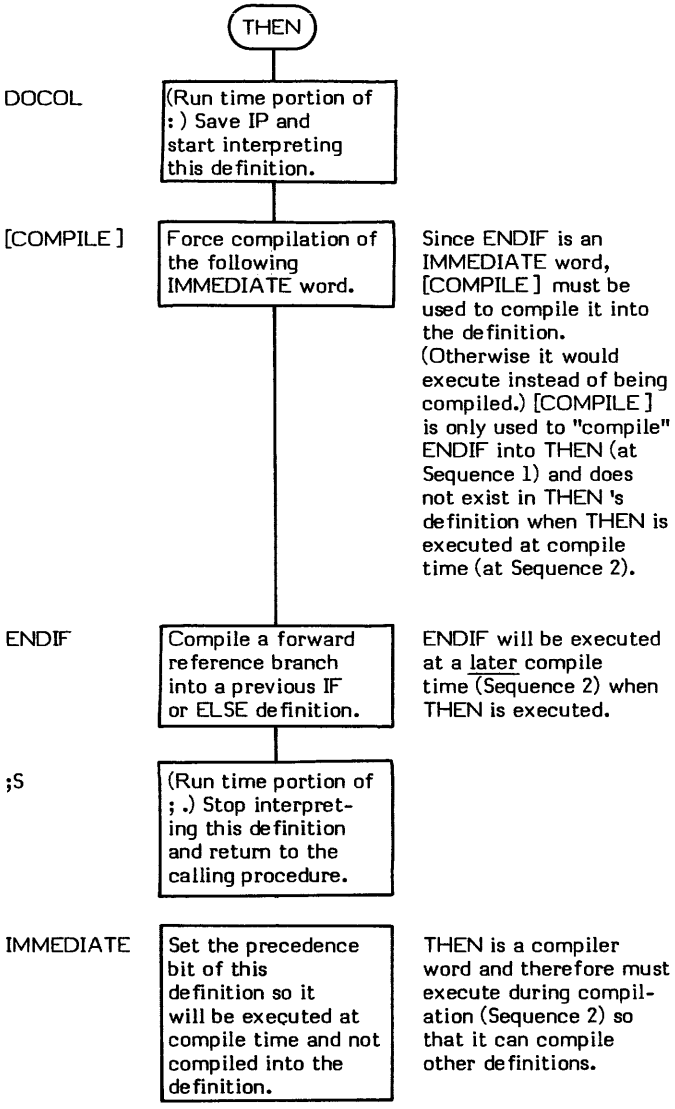
Refer to IF , ELSE , and ENDIF .

FORTH-79: The FORTH-79 equivalent for THEN is THEN .

Definition:

```
:  THEN  ( offset address\ 2 -- )  ( compile time )
     [COMPILE ] ENDIF  ;  IMMEDIATE
```

COMPILE TIME action of THEN (Sequence 1): (offset address\ 2 —)



EXECUTION TIME action of THEN (Sequence 3): (—)

THEN simply serves as the destination of a forward branch from a previous IF or ELSE statement.

TIB (— data address)

TIB (pronounced "T-I-B" for Terminal Input Buffer) is a user variable that contains the address of the Terminal Input Buffer.

The Terminal Input Buffer is an area of memory reserved as a buffer area for the data stream coming from the terminal. The buffer is normally located in memory between the return stack and the parameter stack. The return stack "grows downward" into the buffer.

The length of the buffer is usually 82 (decimal) bytes long, 80 characters + 2 terminators. The length is determined by a literal value in QUERY .

TIB is initialized by COLD during system start up with data from the origin parameter area.

The user variable TIB is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable TIB .

Refer to QUERY , and USER .

FORTH-79: There is no FORTH-79 equivalent for TIB .

TOGGLE

TOGGLE (address \ bit mask —)

TOGGLE Exclusive-OR's the contents of the memory location pointed to by the second stack location with the value on the top of the stack. Note: This is opposite "normal" usage (i.e., the address is usually on the top of the stack and the data is in the second stack entry).

This word is named TOGGLE because of the effect of the Exclusive-OR. "Toggling" a specific bit once, changes the state of the bit. "Toggling" that same bit again changes the bit back to its original state. (Using the same bit mask both times, of course.)

Note that the word operates on one byte in memory.

Also note that word addressing machines may behave differently.

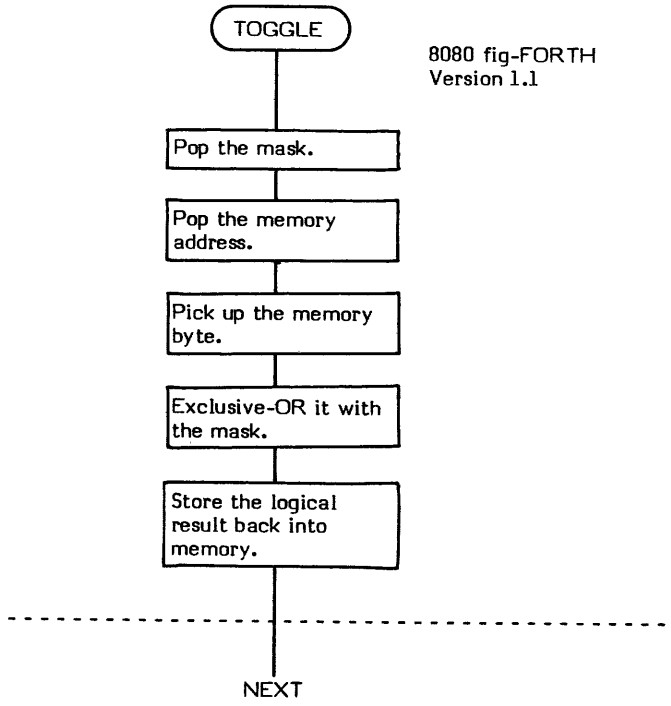
SMUDGE is an example of a word that uses TOGGLE .

* **At entry** - The top of the parameter stack contains a 16-bit value. The low order 8 bits of this value are an Exclusive-OR bit mask. THe high order 8 bits are ignored. The second stack entry contains the memory address of the location to be "toggled".

* **At exit** - No parameters. The specified memory location contains the logical result of the Exclusive-OR.

TOGGLE is a low level code primitive.

FORTH-79: There is no FORTH-79 equivalent for TOGGLE .



TRAVERSE

TRAVERSE (beginning address \ direction -- ending address)

TRAVERSE calculates the address of the opposite end of a fig-FORTH variable length Name Field. Given either the address of the length byte or the address of the last letter of a Name Field, TRAVERSE moves across ("traverses") that Name Field. Note this means TRAVERSE moves across a Name Field in either direction. See the "At entry" section.

TRAVERSE makes use of the fact that the "terminator flag" for both ends of the Name Field is the most significant bit (the 80H bit). By moving across the Name Field looking for a byte whose value is greater than 7FH.

The words NFA and PFA are examples of words which use TRAVERSE .

* **At entry** - The top of the parameter stack contains a 16-bit direction indicator value.

  If this direction indicator is a 1, motion is toward high memory and the second stack entry must be the address of the Name Field's length byte.

  If the direction indicator is a -1, motion is toward low memory and the second stack entry must be the address of the last letter of the Name Field.

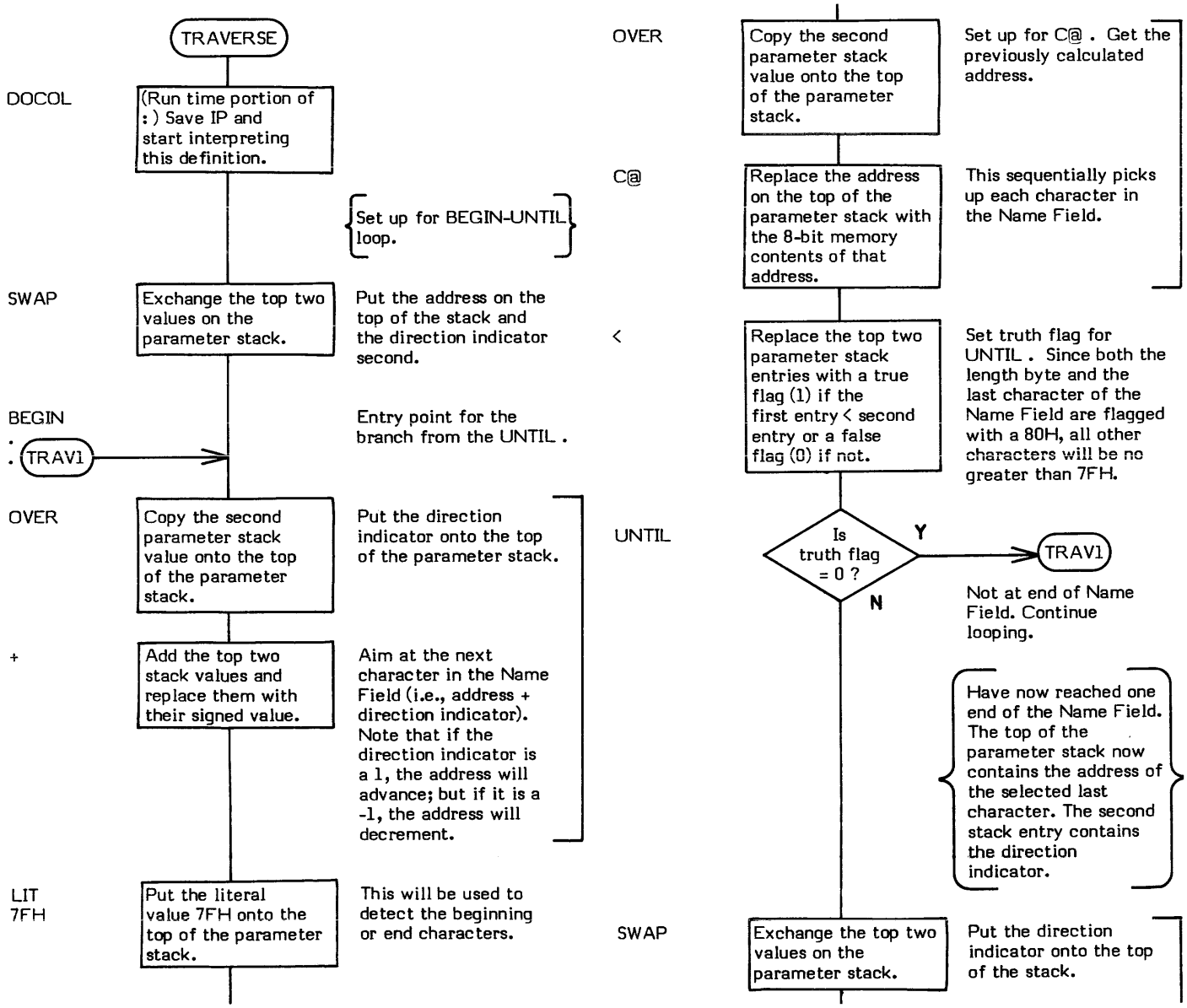
* **At exit** - The top of the parameter stack contains the address of the opposite end of the Name Field. The direction indicator is dropped.

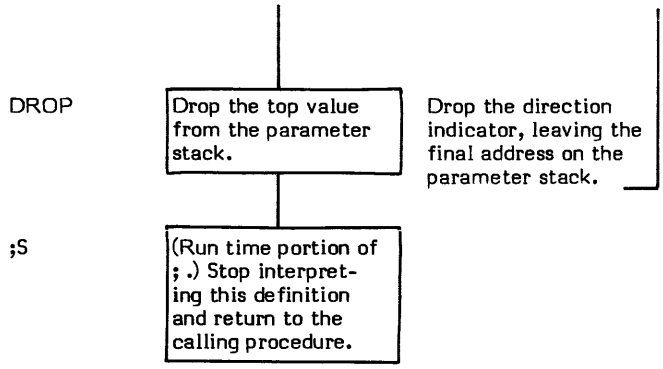
TRAVERSE is a high level colon definition.

FORTH-79: There is no FORTH-79 equivalent for TRAVERSE .

Definition:

```
:  TRAVERSE  ( beginning address \ direction -- ending address )
     SWAP      BEGIN
                    OVER + 7F    OVER C@    <
               UNTIL
     SWAP    DROP    ;
```




DROP
Drop the top value from the parameter stack.
Drop the direction indicator, leaving the final address on the parameter stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

TRIAD (Screen number —)

TRIAD outputs three screens to the output device, i.e., one page. The page will begin with a screen whose number is divisible by three. One of the screens on the page will be the screen specified.

A message, taken from Line 15 of Screen 4, is listed on the bottom of the page.

In the 8080 fig-FORTH Version 1.1, pressing any terminal key will terminate the listing.

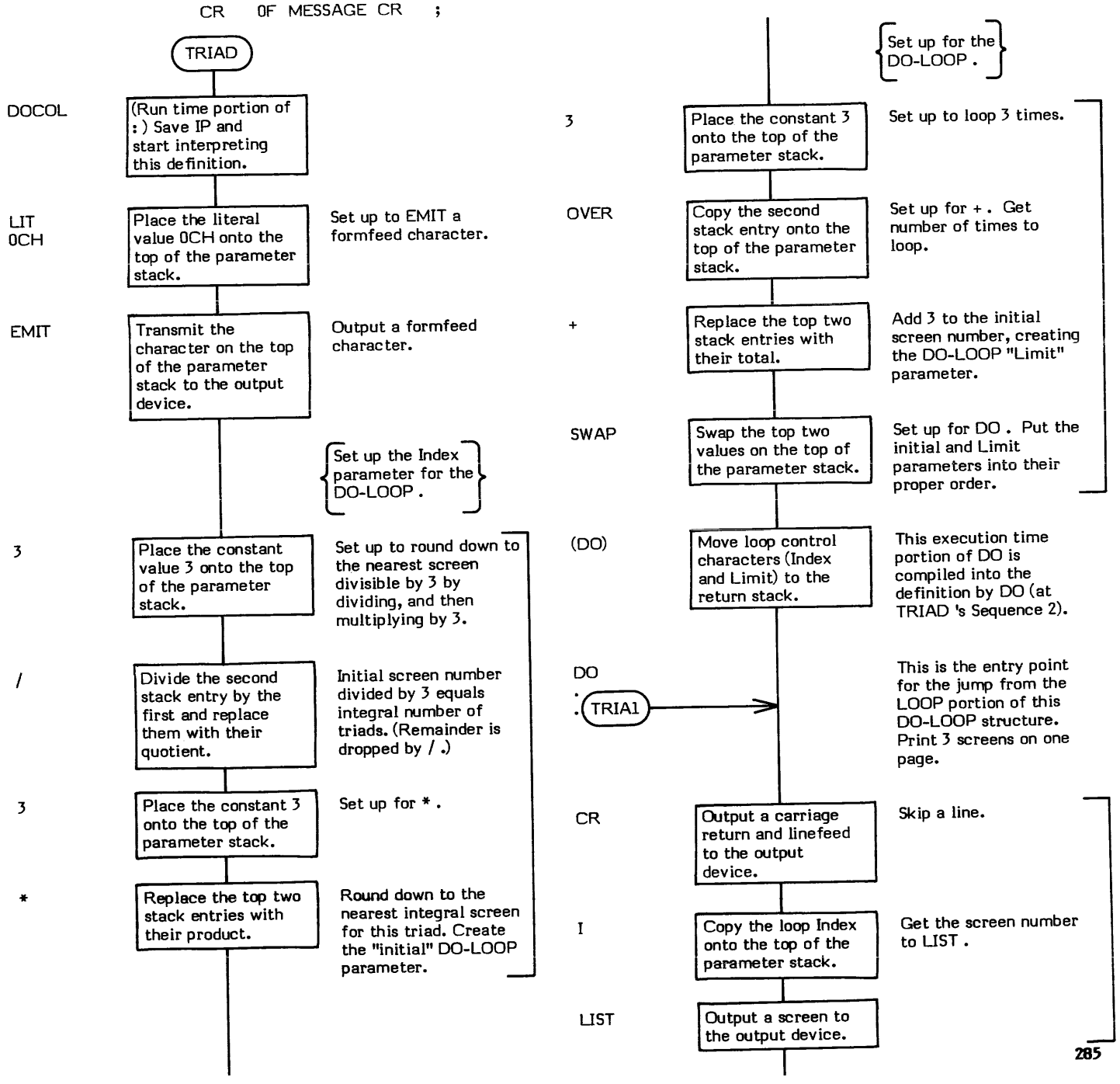
* **At entry** - The top of the parameter stack contains a 16-bit unsigned number specifying one of the screens to be listed.
* **At exit** - No parameters.

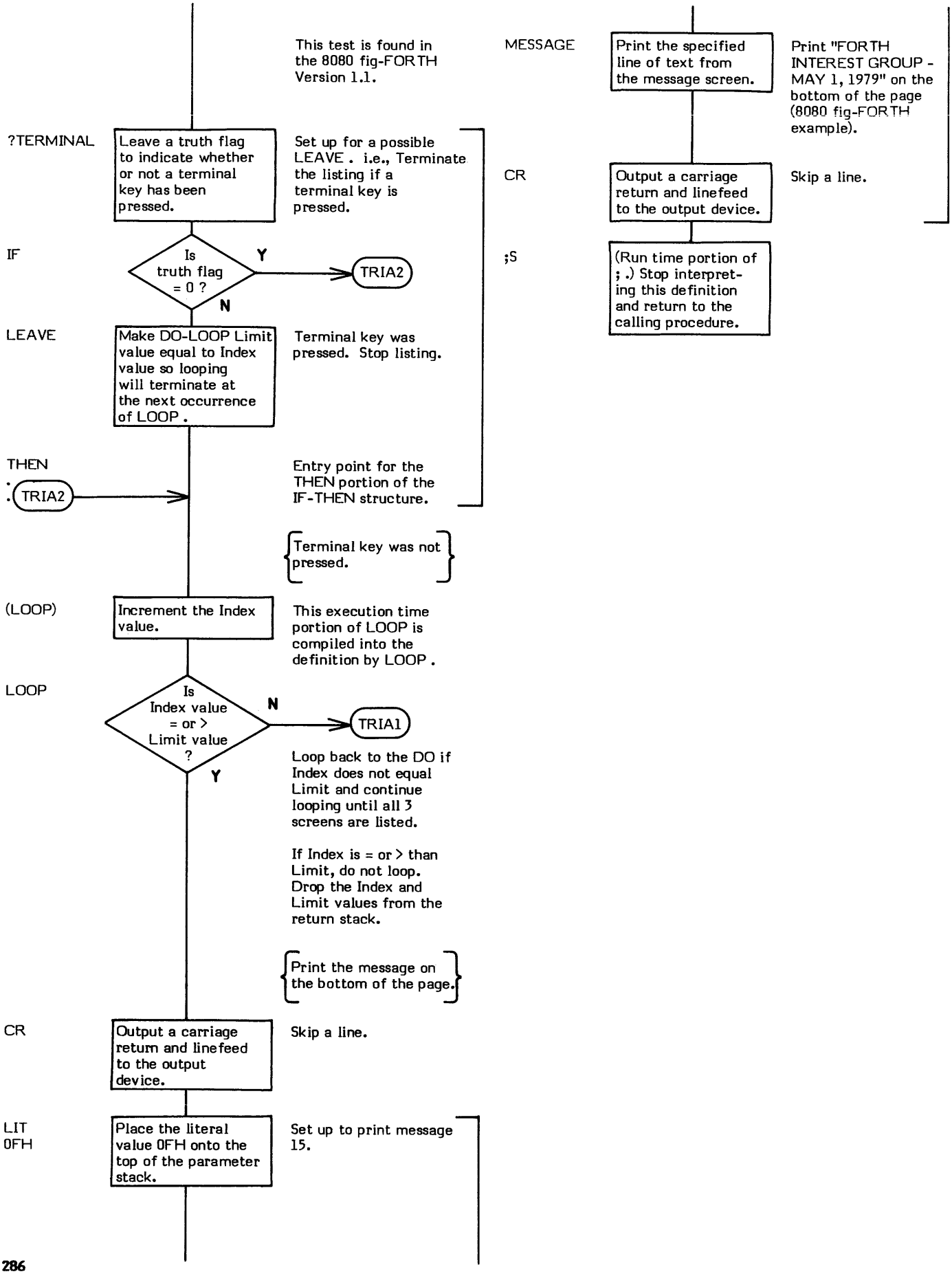
TRIAD is a high level colon definition.

FORTH-79: There is no FORTH-79 equivalent for TRIAD .

Definition:

```
: TRIAD ( # - ) ( 8080 Version 1.1 )
    0C EMIT  3 / 3 *  3 OVER + SWAP
    DO
        CR I LIST  ?TERMINAL  IF  LEAVE  THEN
    LOOP
    CR  0F MESSAGE CR  ;
```




This test is found in the 8080 fig-FORTH Version 1.1.
?TERMINAL
Leave a truth flag to indicate whether or not a terminal key has been pressed.
Set up for a possible LEAVE . i.e., Terminate the listing if a terminal key is pressed.
IF
Is truth flag = 0 ?
Y
TRIA2
N
LEAVE
Make DO-LOOP Limit value equal to Index value so looping will terminate at the next occurrence of LOOP .
Terminal key was pressed. Stop listing.
THEN
TRIA2
Entry point for the THEN portion of the IF-THEN structure.
Terminal key was not pressed.
(LOOP)
Increment the Index value.
This execution time portion of LOOP is compiled into the definition by LOOP .
LOOP
Is Index value = or > Limit value ?
N
TRIA1
Y
Loop back to the DO if Index does not equal Limit and continue looping until all 3 screens are listed.
If Index is = or > than Limit, do not loop. Drop the Index and Limit values from the return stack.
Print the message on the bottom of the page.
CR
Output a carriage return and linefeed to the output device.
Skip a line.
LIT 0FH
Place the literal value 0FH onto the top of the parameter stack.
Set up to print message 15.
MESSAGE
Print the specified line of text from the message screen.
Print "FORTH INTEREST GROUP - MAY 1, 1979" on the bottom of the page (8080 fig-FORTH example).
CR
Output a carriage return and linefeed to the output device.
Skip a line.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

TYPE **(beginning address \ number of characters —)**

TYPE outputs a character string to the output device. No output takes place if the supplied length is zero.

D.R is an example of a word that uses TYPE .

* **At entry** - The top of the parameter stack contains an absolute 16-bit single precision number of characters to be output. The second entry contains the 16-bit beginning address of the characters to be typed.

* **At exit** - No parameters.

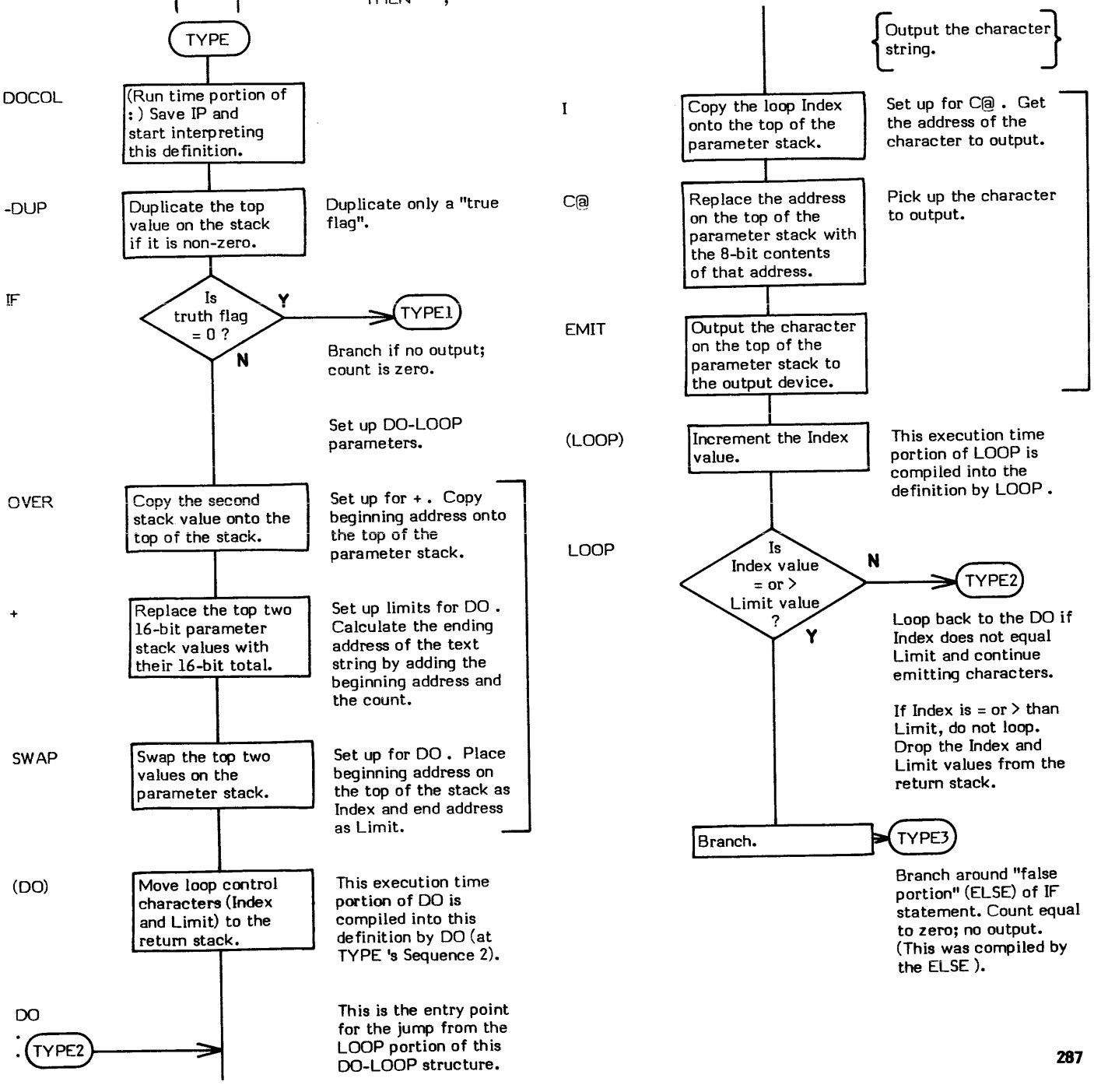
TYPE is a high level colon definition.

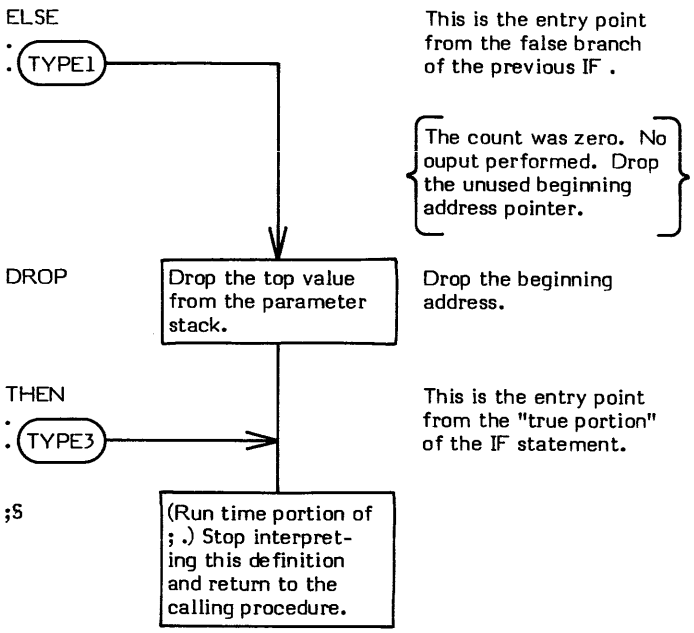
Refer to COUNT .

FORTH-79: The FORTH-79 equivalent for TYPE is TYPE .

Definition:

```
: TYPE ( beginning address\ number of characters -- )
   -DUP IF
      OVER + SWAP DO
         I C@ EMIT
      LOOP
   ELSE DROP
   THEN ;
```




ELSE
TYPE1
This is the entry point from the false branch of the previous IF .
The count was zero. No ouput performed. Drop the unused beginning address pointer.
DROP
Drop the top value from the parameter stack.
Drop the beginning address.
THEN
TYPE3
This is the entry point from the "true portion" of the IF statement.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

U* **(unsigned single multiplier \ unsigned single multiplicand — double unsigned product)**

U* (pronounced "U-star") is an unsigned multiplier. It inputs two unsigned 16-bit values and returns an unsigned 32-bit product.

The heart of U* (on a processor without a multiply instruction) is a multiply routine which works on the standard "shift and add algorithm". Two partial products are derived which are then added together to obtain a 32-bit unsigned product. Note that U* is unsigned. M* should be used if a signed product is desired.

M* is an example of a word which uses U* .

WARNING -- fig-FORTH U* actually does not multiply unsigned 16-bit values. It only correctly multiplies signed positive 16-bit values.

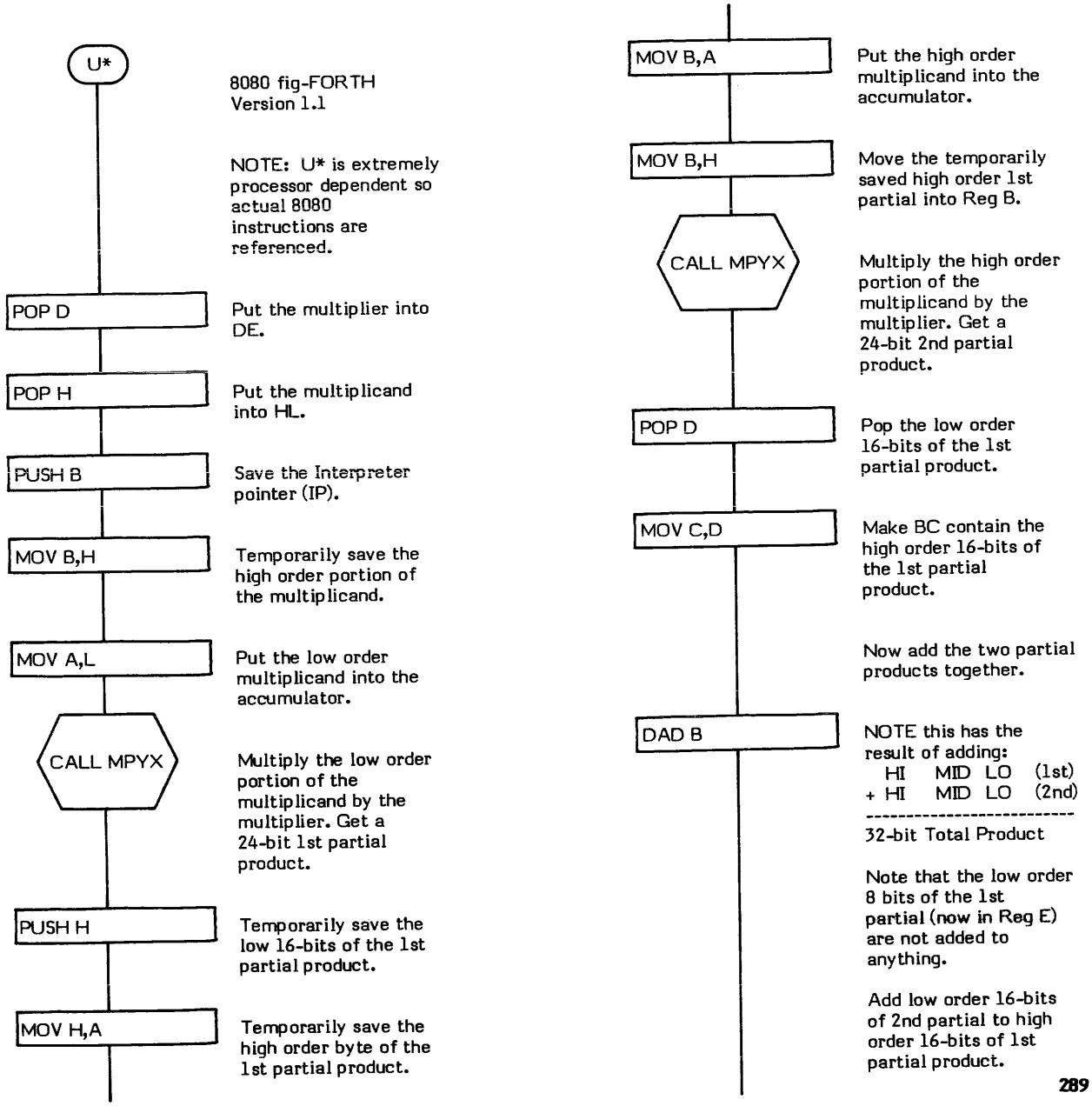
* **At entry** - The top of the parameter stack contains a 16-bit unsigned single precision multiplier. The second stack entry contains a 16-bit unsigned single precision multiplicand.

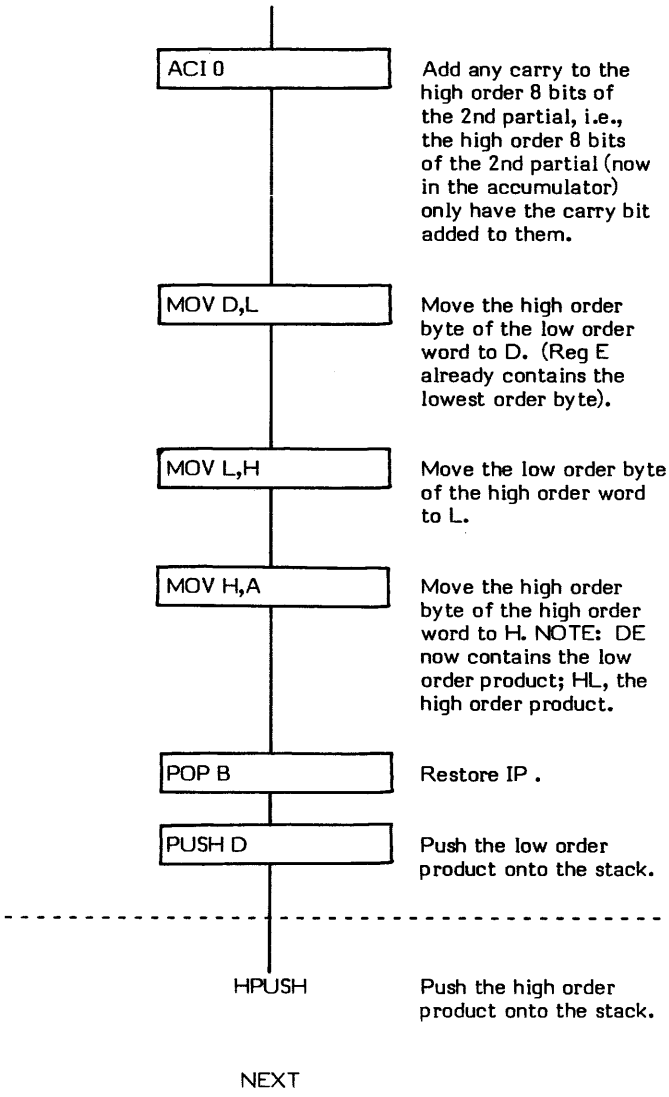
* **At exit** - The top of the parameter stack contains the high order 16-bits of an unsigned 32-bit double precision product. The second stack entry contains the low order 16-bits of the 32-bit product.

U* is a low level code primitive.

Refer to M* .

FORTH-79: The FORTH-79 equivalent for U* is U* .

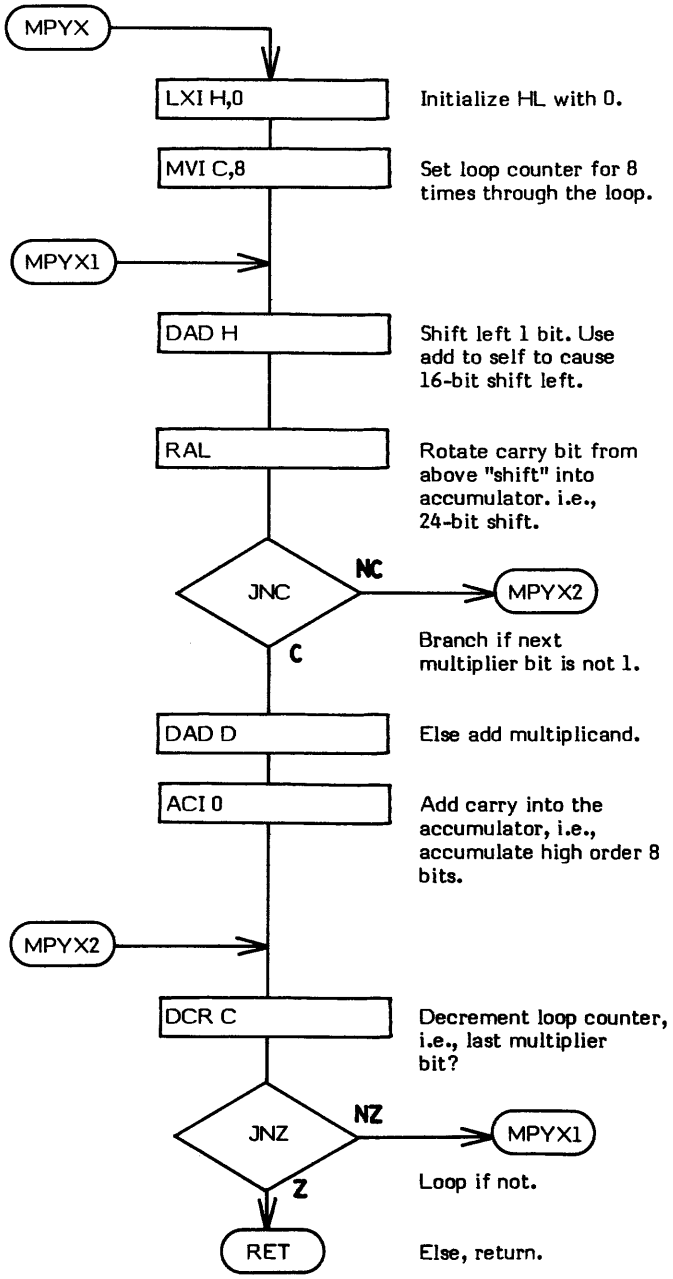




Multiply subroutine MPXY used by U*.

Given: A - Multiplier
DE - Multiplicand

Product returned in: A - High order 8 bits
HL - Low order 16 bits



U. **(value --)**

U. (pronounced "U-dot") performs an unsigned binary-to-ascii conversion on the 16-bit value on the top of the stack and prints the result on the output device followed by one space.

The current value in BASE is used as the conversion radix.

U. has the same effect as . but the output is not signed. This is beneficial when displaying 16-bit addresses which have their high order bit set, since . prints all addresses above 7FFFH as negative numbers.

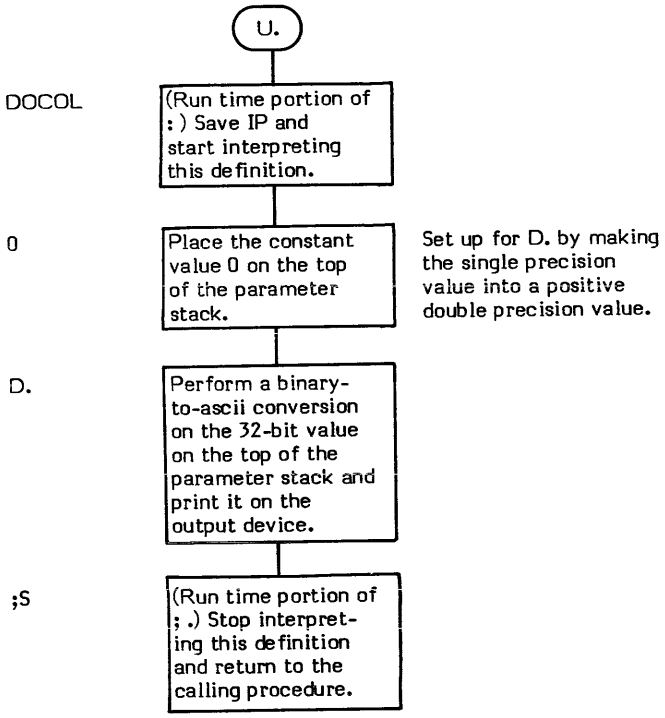
* **At entry** - The top of the parameter stack contains a 16-bit value to be converted and printed.
* **At exit** - No parameters.

U. is a high level colon definition.

Refer to . (dot).

FORTH-79: The FORTH-79 equivalent for U. is U. .

Definition:
```
:  U.  ( value -- )
      0  D.  ;
```



U/

U/ (unsigned double precision dividend \ unsigned divisor — unsigned remainder \ unsigned quotient)

U/ (pronounced "U-slash") performs unsigned division upon a double precision dividend using a single precision divisor. U/ is the primitive division word used by all of the other division words such as M/ , MOD , and M/MOD . Note that U/ performs unsigned division.

U/ is a low level primitive that executes in the machine code of the host processor. For those processors which have a divide instruction, U/ is very straightforward. For processors without a divide instruction, implementing a divide is not a trivial task.

Division on processors which do not have a divide instruction is usually performed via the "shift and subtract" algorithm. This algorithm is conceptually very similiar to performing long division by hand. An attempt is made to divide the divisor into the highest order dividend column. If the dividend is larger than the divisor, division takes place and a value (1-9 for decimal long division, 1 for binary division) is placed in the corresponding quotient column. If division cannot take place, a 0 is placed in the corresponding quotient column. Division of the next dividend column then takes place including any remainder from the previous column's division. This continues until all columns have been divided.

The 8080 implementation of this algorithm can most easily be understood by first looking at the following simplified example.

In the example in Figure U/-1, 41 (the dividend) is divided by 3 (the divisor).

Let us envision that the division is taking place on a simplistic processor that has only two registers. At the start, the high order portion of the binary 41 is in Register X and the low order portion is in Register Y.

Since our processor only has two registers, we will develop the quotient in the low order register as the dividend is shifted out. This is perfectly acceptable since a dividend column (bit) is shifted out to the left while a corresponding quotient column is shifted in from the right.

When the division is complete, Register Y contains the quotient and Register X contains any remainder.

Our processor can subtract 4 bits at a time from the high order portion of the dividend (Register X). Therefore, in order to divide the whole dividend, we will have to shift and subtract both registers 4 times to completely divide both the high and low portions of the dividend.

The dividend is 41 (decimal) or 29 (hex) or 00101001 (binary). At the start, Register X contains the high order portion of "binary 41" (0010) and Register Y, the low order portion (1001).

After the first shift left, Register X contains a 5 (0101) and a 0 quotient bit has been shifted into the least significant bit (LSB) of Register Y. The divisor, 3, (0011) is then subtracted from Register X, leaving a remainder of 2 (0010). Since a subtraction was possible, the quotient bit is changed from 0 to 1.

Ater the second shift left, Register X contains a 4 (0100) and a 0 quotient bit has been shifted into the LSB of Register Y. The divisor, 3 (0011), is then subtracted from Register X, leaving a remainder of 1 (0001). Since a subtraction was possible, the quotient bit is changed from 0 to 1.

After the third shift left, Register X contains a 2 (0010) and another 0 quotient bit has been shifted into the LSB of Register Y. An attempt is then made to subtract the divisior, 3 (0011), from Register X; but since 3 is larger than 2, a subtraction cannot be made. (NOTE: In the actual algorithm, the subtraction is always performed and if underflow is detected, the divisor value is added back to the dividend.) The value 2 (0010) is left in Register X and the corresponding quotient bit in Register Y remains a 0.

After the fourth shift left, Register X contains a 5 (0101) and another 0 quotient bit has been shifted into the LSB of Register Y. The divisior, 3 (0011) is again subtracted from Register X, leaving a remainder of 2 (0010). Since a subtraction was possible, the quotient bit is changed from 0 to 1.

Since four shifts have occurred, the division is complete. Register Y contains the quotient, 13 decimal (1101); and Register X contains the remainder 2 decimal (0010) -- which is the correct answer obtained by dividing decimal 41 by 3.

	REG X	REG Y	
START	0 0 1 0	1 0 0 1	41 / 3 = 13 rem 2
1	0 1 0 1	0 0 1 0	
	1 1		
	0 0 1 0	0 0 1 1	
2	0 1 0 0	0 1 1 0	
	1 1		
	0 0 0 1	0 1 1 1	
3	0 0 1 0	1 1 1 0	
	1 1		
4	0 1 0 1	1 1 0 0	
	1 1		
	0 0 1 0	1 1 0 1	

Figure U/-1

Example of "Shift-Subtract" Division Algorithm

Once the example is understood, the actual 8080 implementation of the algorithm is also understandable. Registers DE contain the low order dividend. This corresponds to Register Y in the example. Registers HL contain the high order portion of the dividend. This corresponds to Register X. As in the example, registers DE will contain the quotient and registers HL will contain the remainder. The divisor is kept in registers BC.

The register pair DE is 16-bits wide, therefore 16 shifts will be performed.

When the routine is first entered, a check is made to determine if the divisor is larger than the dividend. If so, both remainder and quotient are set to FF (hex) and the routine is exited.

Otherwise, a loop counter is initialized to shift 16 times and the shift-subtract sequence begins. A 32-bit double precision shift is performed between registers DE and HL. If a carry bit is shifted out, it is known absolutely that a subtraction could occur without true underflow so the divisor (BC) is subtracted from the high order dividend (HL). (Any underflow in this case would simply be a remainder with a borrow occurring from the carry.) If a carry bit is not shifted out, an underflow occurrence would mean that subtraction could not take place, so a separate subtraction routine is executed (although the identical function is performed, i.e., the divisor is subtracted from the high order dividend). The difference with this routine is that it checks for underflow and if detected adds the previously subtracted divisor back to the dividend. The quotient value is decremented by 1 so that when the automatic quotient increment is executed it will be the same as if no increment were performed. (NOTE: This decrement is just a programming "trick" and has nothing really to do with the algorithm. The intent is to set the quotient bit to 1 for a successful subtraction or to 0 for an un-successful subtraction.)

This shift-subtract is performed 16 times. At that time, the quotient is in register DE and the remainder is in HL.

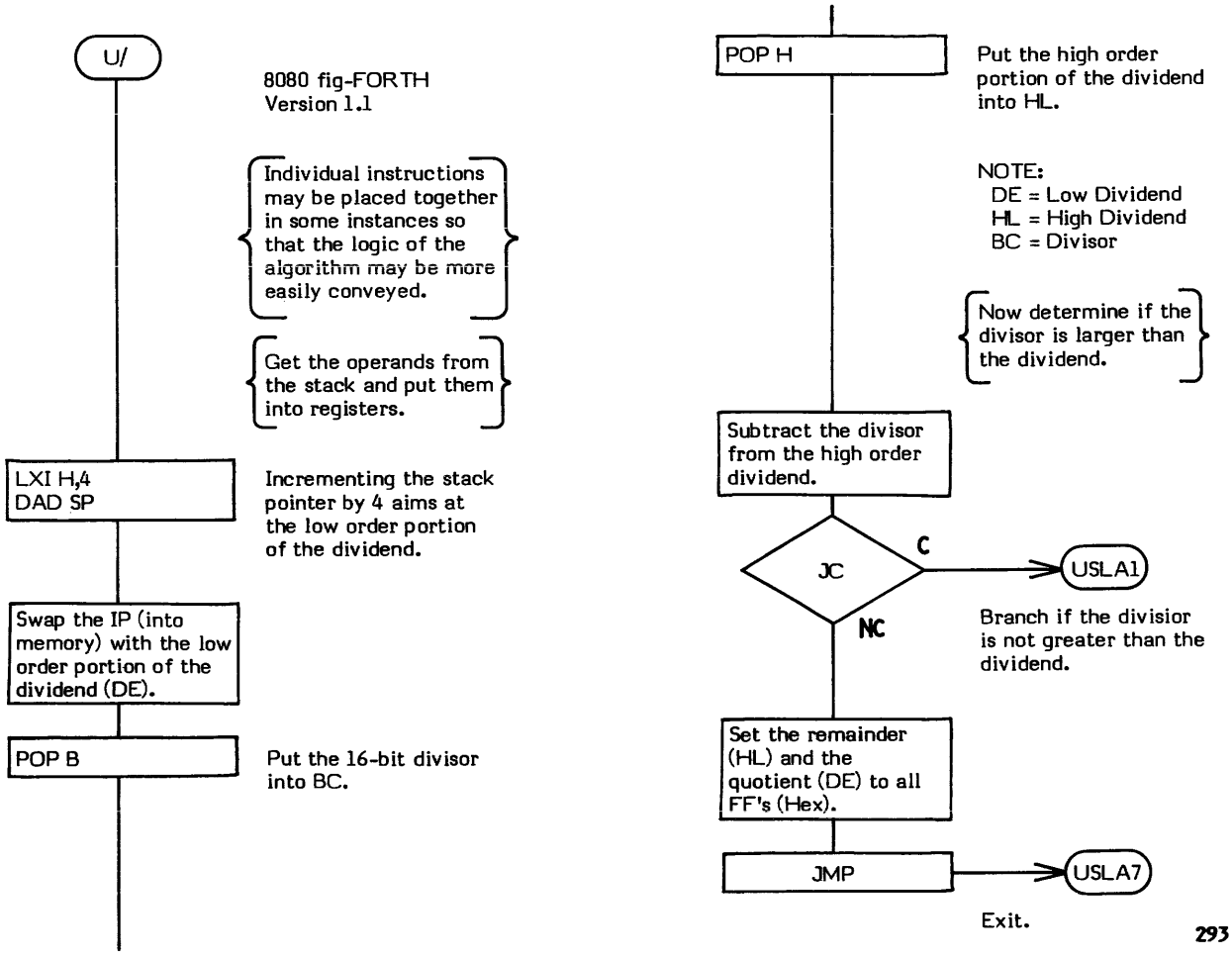
* **At entry** - The top of the parameter stack contains an unsigned 16-bit single precision divisor. The second and third stack entries contain an unsigned 32-bit double precision dividend. (Note that some fig-FORTH implementations require a 31-bit positive signed vlaue.) The second stack entry contains the high order portion of the dividend; the third, the low order portion.

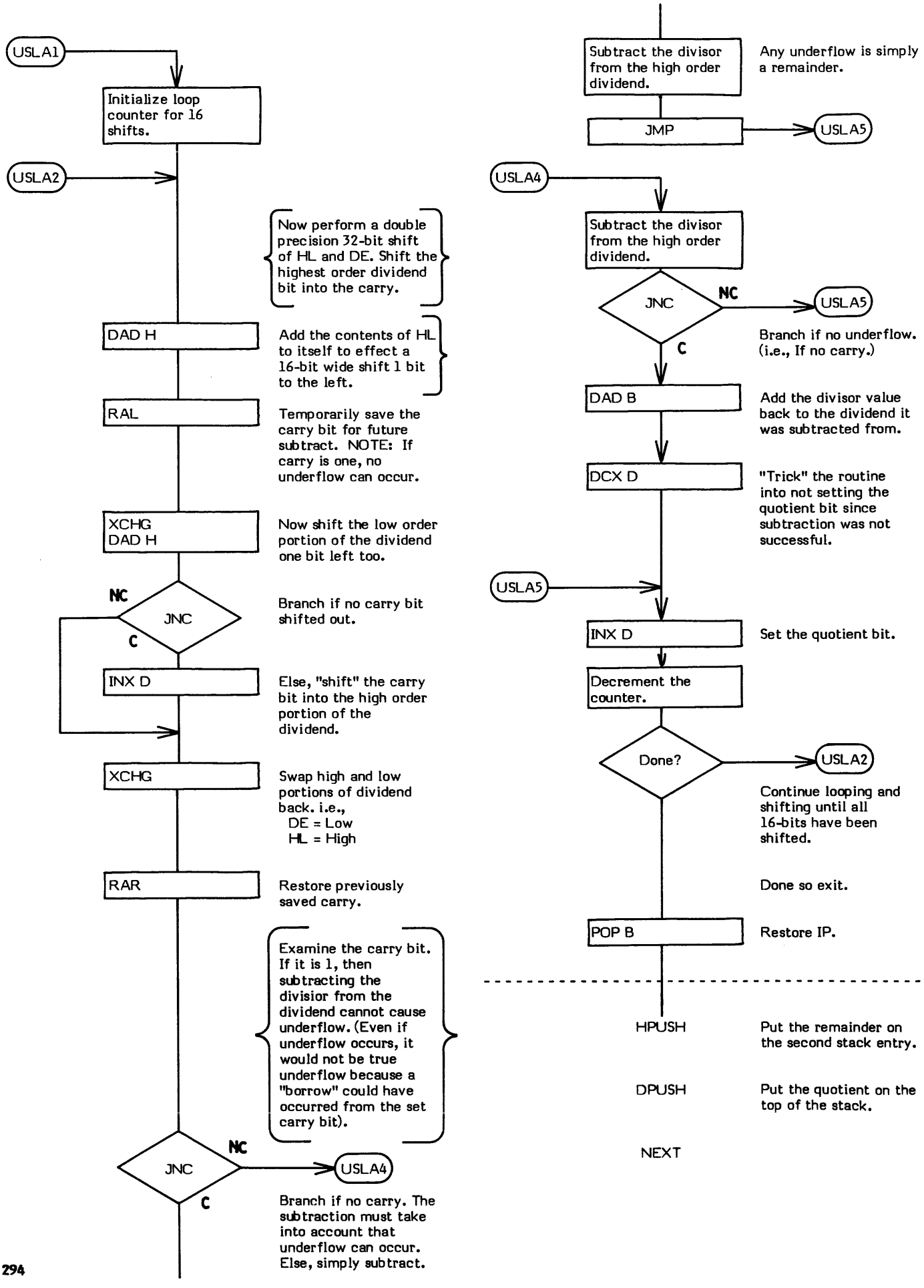
* **At exit** - The top of the parameter stack contains an unsigned 16-bit single precision quotient. The second stack entry contains the unsigned 16-bit remainder from the division.

U/ is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for U/ is U/MOD .

Note that this is an extremely processor-dependent routine and therefore the flowchart often references the 8080 instruction set.




USLA1
Initialize loop counter for 16 shifts.
USLA2
Now perform a double precision 32-bit shift of HL and DE. Shift the highest order dividend bit into the carry.
DAD H
Add the contents of HL to itself to effect a 16-bit wide shift 1 bit to the left.
RAL
Temporarily save the carry bit for future subtract. NOTE: If carry is one, no underflow can occur.
XCHG
DAD H
Now shift the low order portion of the dividend one bit left too.
NC
JNC
C
Branch if no carry bit shifted out.
INX D
Else, "shift" the carry bit into the high order portion of the dividend.
XCHG
Swap high and low portions of dividend back. i.e.,
DE = Low
HL = High
RAR
Restore previously saved carry.
Examine the carry bit. If it is 1, then subtracting the divisior from the dividend cannot cause underflow. (Even if underflow occurs, it would not be true underflow because a "borrow" could have occurred from the set carry bit).
NC
JNC
USLA4
C
Branch if no carry. The subtraction must take into account that underflow can occur. Else, simply subtract.
Subtract the divisor from the high order dividend.
Any underflow is simply a remainder.
JMP
USLA5
USLA4
Subtract the divisor from the high order dividend.
NC
JNC
USLA5
C
Branch if no underflow. (i.e., If no carry.)
DAD B
Add the divisor value back to the dividend it was subtracted from.
DCX D
"Trick" the routine into not setting the quotient bit since subtraction was not successful.
USLA5
INX D
Set the quotient bit.
Decrement the counter.
Done?
USLA2
Continue looping and shifting until all 16-bits have been shifted.
Done so exit.
POP B
Restore IP.
HPUSH
Put the remainder on the second stack entry.
DPUSH
Put the quotient on the top of the stack.
NEXT

UNTIL

COMPILE TIME (Sequence 2): (loop address\ 1 —)

EXECUTION TIME (Sequence 3): (truth flag —)

UNTIL is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

UNTIL is used to mark the end of an indefinite, conditional loop structure where repetition continues "until" the boolean input to UNTIL is true.

UNTIL is used in the form:

BEGIN "Loop Body" UNTIL

(END may be used in place of UNTIL but UNTIL is the preferred word.)

BEGIN-UNTIL structures must always be used within a colon definition.

It is important to note that in using a BEGIN-UNTIL structure, the "loop body" will always be executed at least once. This is because the exit condition is not tested "until" after the "loop body" has been executed. This is known as a "post-test" loop. If the exit condition must be tested before "loop body" execution, the BEGIN-WHILE-REPEAT structure should be used instead.

The compile time action of UNTIL is to compile a 0BRANCH into the dictionary. Secondly, it resolves the "loop body" entry point address provided by BEGIN into a return branch offset used by the 0BRANCH and stores this offset into the definition.

Some compiler security is provided by checking for a 1 on the top of the stack. An UNTIL without a preceding BEGIN will probably not encounter a 1 on the top of the stack. (During compilation, BEGIN leaves a 1 on the top of the stack.)

The execution time action (Sequence 3) of UNTIL is to provide a conditional repetitive branch back to the loop's corresponding BEGIN (i.e., the beginning of the "loop body"). The input parameter to BEGIN is a boolean flag. If this flag is false (0), control returns to the first word in the "loop body" (just after BEGIN). If the boolean flag is true (not 0), no branch occurs and the loop is exited. That is, repetitive looping continues "until" the exit conditional is true.

The 0BRANCH, compiled into the definition at compile time, is what controls the looping.

Note that UNTIL is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

BLOCK is an example of a word which uses UNTIL .

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 1 used for compiler security. The second stack entry contains the 16-bit entry point address of the "loop body" portion of the BEGIN-UNTIL structure.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains a 16-bit signed single precision boolean flag used to control the conditional looping of the structure.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) — This word may only be used within a colon definition.

CONDITIONALS NOT PAIRED (13H) — There is some sort of problem with the pairing of conditionals within the definition being compiled.

UNTIL is a high level colon definition.

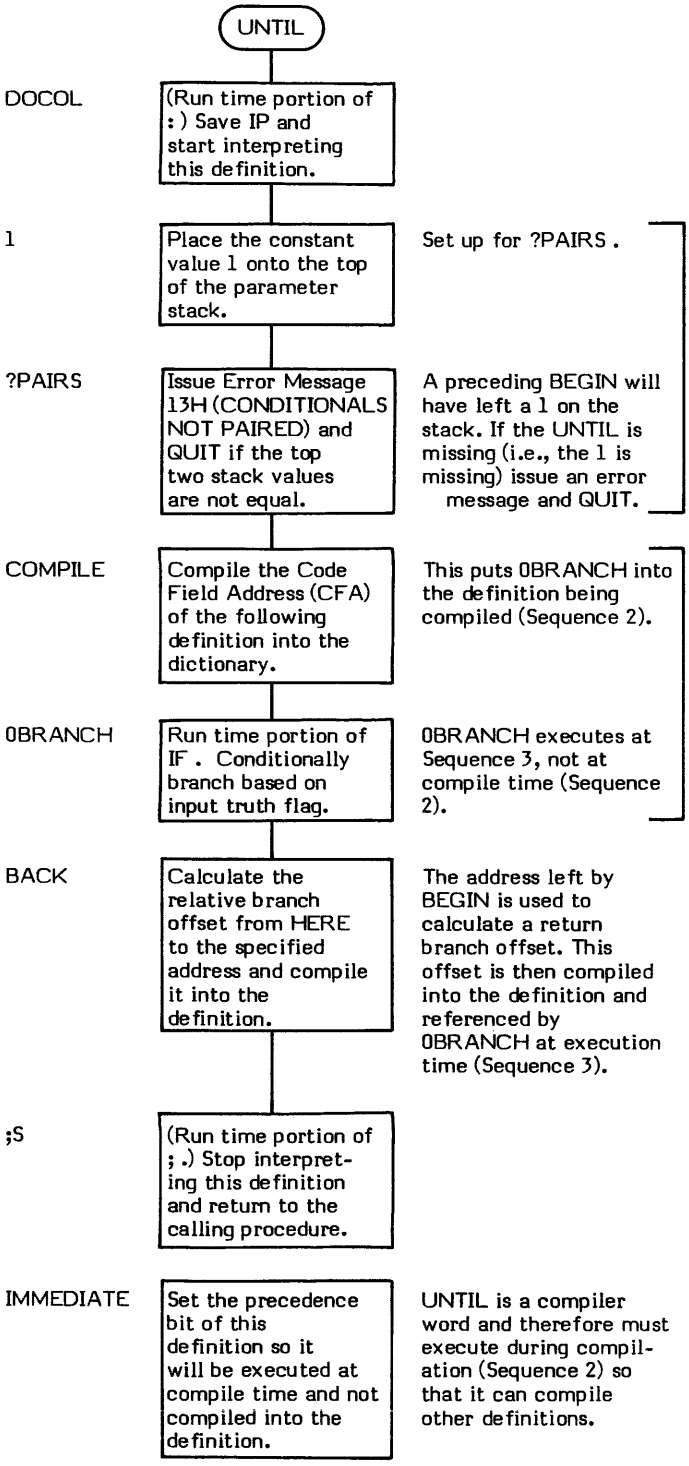
Refer to BEGIN , 0BRANCH , and END .

FORTH-79: The FORTH-79 equivalent for UNTIL is UNTIL .

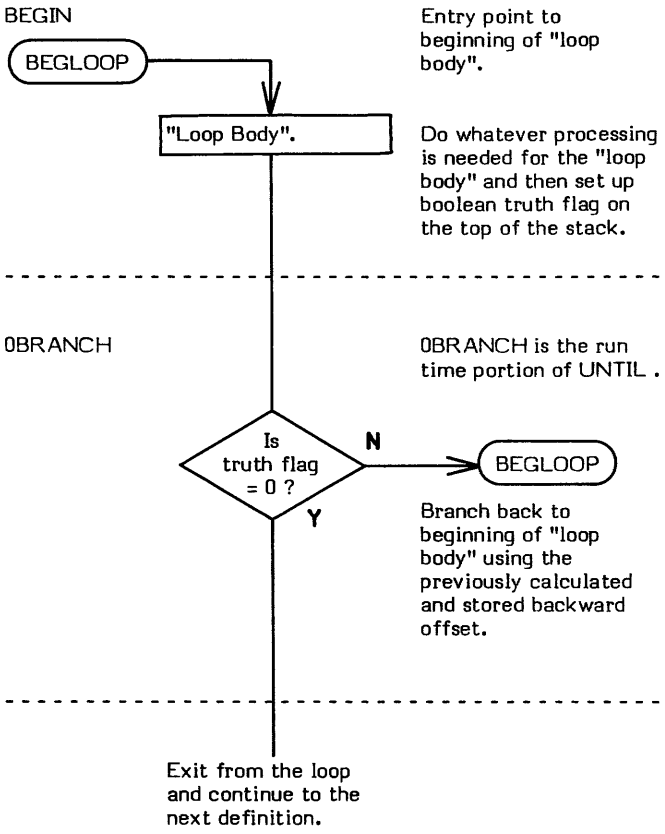
Definition:

```
:  UNTIL  ( loop address\ 1 -- )  ( compile time )
       1 ?PAIRS  COMPILE 0BRANCH  BACK  ;  IMMEDIATE
```

COMPILE TIME action of UNTIL : (loop address \ 1 —)
(Sequence 2)



EXECUTION TIME action of UNTIL : (truth flag —)
(Sequence3)



UPDATE (--)

UPDATE flags the most-recently-accessed buffer so that the buffer's data will be written to mass storage when that buffer is re-allocated or the word FLUSH is executed. The update bit (or flag) is the most significant bit (MSB) of the header. (i.e., The block number portion of a buffer.) Refer to +BUF for a detailed description of the buffers and their organization.

The intent of UPDATE is to flag the data contained within a buffer as being changed or modified. Note, however, that it is perfectly legal to UPDATE a buffer that has not been modified although this action may cause an unnecessary write of information that is identical to that already on the disk.

It is extremely important to note, that the simple act of modifying data in a buffer will not cause it to be written back to disk. The update flag must be set in order for this to occur.

It is also equally important to note that setting the update flag does not absolutely guarantee that data will be written to disk. This write only occurs when a buffer is allocated. If the system is restarted, or powered off, or the desired disk is removed from the drive; the data will not be written to the desired location. This problem is easily solved by using the word FLUSH to force all updated buffers to be written to disk before allowing any of the conditions mentioned above to occur.

UPDATE should always be executed at the end of a sequence which modifies data in a buffer. "Updating" a buffer more than once does no harm and takes very little execution time; especially in comparison to writing the data to disk each time a change takes place.

* **At entry** - No parameters.
* **At exit** - No parameters.

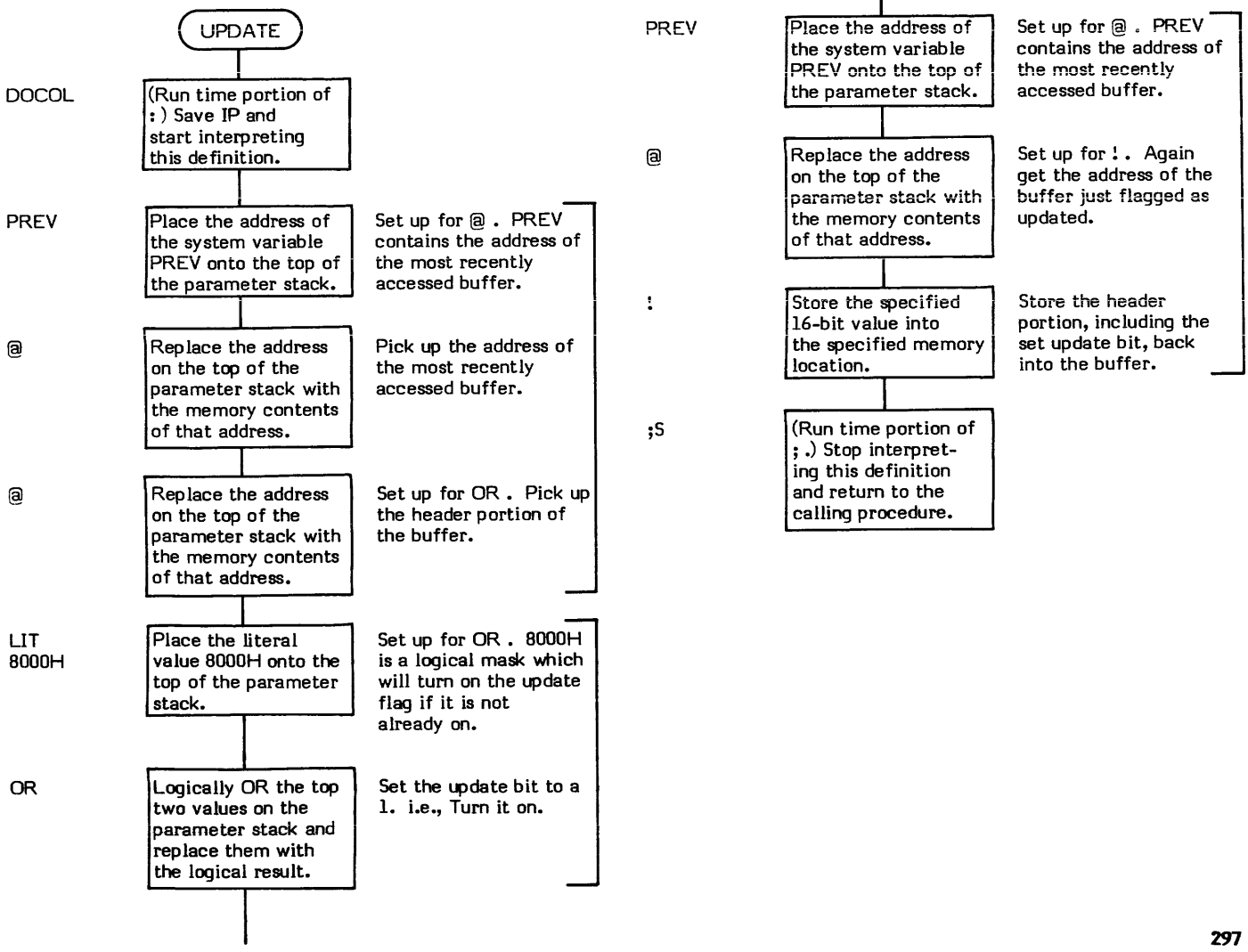
UPDATE is a high level colon definition.

Refer to +BUF , BLOCK , USE , EMPTY-BUFFERS , and FLUSH .

FORTH-79: The FORTH-79 equivalent for UPDATE is UPDATE .

Definition:

```
: UPDATE ( -- )
    PREV @ @  8000 OR  PREV @ !  ;
```



USE

USE (— data address)

USE is a system variable which contains the address of the buffer to "use" next. USE is referenced by buffer allocation routines.

USE is described in detail in the description of BUFFER and +BUF . (Note that USE is a system variable and not a user variable.)

In the fig-Model, USE is normally initialized to point to FIRST (the first buffer). This is done by using the "initial value" feature of the word VARIABLE (e.g., BUF1 VARIABLE USE). Although this technique works, it is not good programming practice and in the 8080 fig-FORTH Version 1.1 USE is initialized by COLD .

The system variable USE is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the system variable USE .

Refer to BUFFER , +BUF , BLOCK , and VARIABLE .

FORTH-79: There is no FORTH-79 equivalent for USE .

USER

COMPILE TIME (Sequence 2): (offset --)

EXECUTION TIME (Sequence 3): (-- address)

USER is a defining word and therefore exhibits two different sets of actions; those at compile time and those at run time.

USER 's compile time (Sequence 2) action is to define a user variable. A user variable is a 16-bit fixed offset (relative to the user pointer, UP) for this user variable. (The value of the offset may not be greater than 255 decimal.)

The form of a defined user variable is:

n USER cccc

where n is the variable's offset from the beginning of the user area and cccc is the name of the variable.

The compile time result of USER is a definition structured like this:

Name Field
Link Field
Code Field - Run Time Code of USER
Parameter Field - Offset Value

USER variables play a very important role in multitasking FORTH systems; however, in non-multitasking systems they could simply be replaced by variables.

Every task in a multitasking environment has functions that are common with other tasks. That is, the system must keep track of each task's base (BASE), its input text buffer pointers (IN), its compilation state (STATE), etc. The simplest way to solve this problem is to:

1. Write routines which perform specific functions for any task.
2. Make these routines access task specific data via a name (i.e., a user variable).
3. Make the execution time code (Sequence 3) for a user variable use the contents of the user area pointer (UP) as a base to which the user variable offset is added.
4. Assign each task a separate user variable area and simply "aim" the user area pointer at the user area of whatever task is active.

TASK 1	TASK 2	TASK 3
User Variable Area 1	User Variable Area 2	User Variable Area 3

UP is changed to point to the active task's user area.

User variables are usually initialized at system start up time by COLD with initialization data stored in the Origin Parameter Area. (Refer to COLD).

The execution time (Sequence 3) action, when the named variable is referenced, is to place the absolute address (offset + beginning address) of the 16-bit variable location onto the top of the parameter stack.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains a 16-bit offset value.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the absolute address of the user variable location.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) -- The dictionary has grown into the Terminal Input Buffer.

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

USER is a high level colon definition.

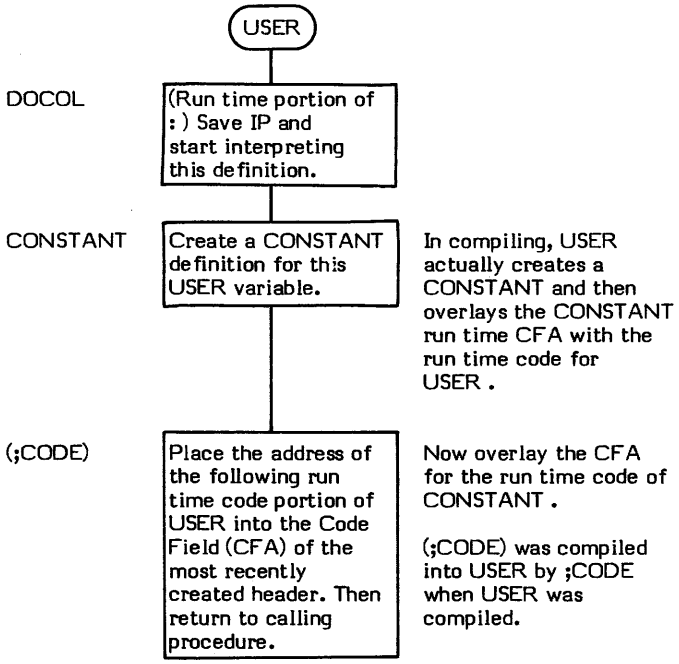
FORTH-79: USER is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Definition:

```
: USER ( offset -- ) ( compile time )
      CONSTANT ;CODE
```

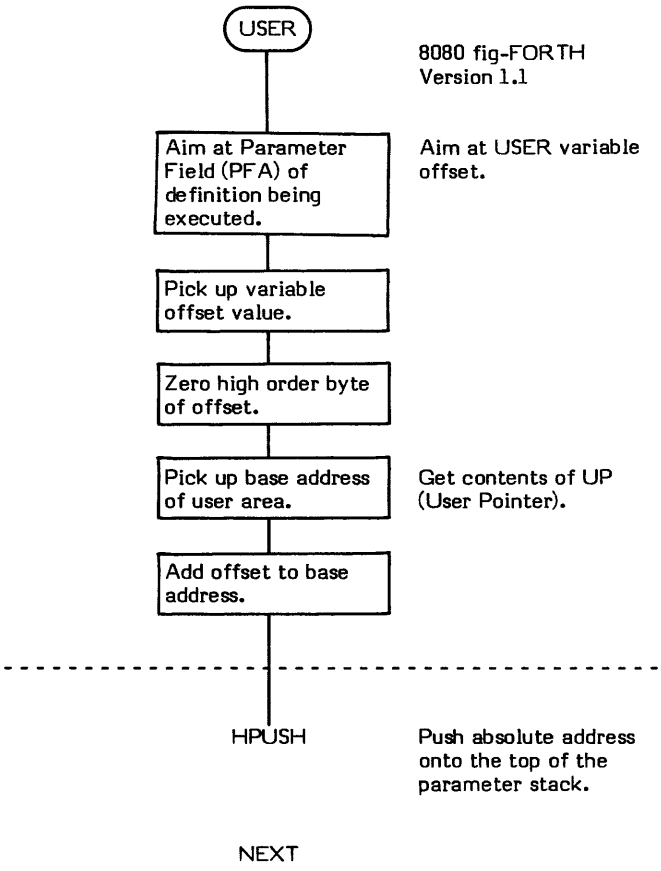
Note: The assembly language execution time code follows the ;CODE .

COMPILE TIME action of USER (Sequence 2): (offset —)



EXECUTION TIME action of USER (Sequence 3): (— address)

Note: This code physically follows ;CODE in the source code for USER.



VARIABLE

COMPILE TIME (Sequence 2): (n —)

EXECUTION TIME (Sequence 3): (— address)

VARIABLE is a defining word and therefore exhibits two different sets of actions; those at compile time and those at execution time.

A VARIABLE in FORTH has the same effect as a variable in most other computer languages. That is, a label is assigned to a specific memory location. Any future references to that location can then be performed via the assigned name. VARIABLE does differ from other languages in that it is active. The equating of a name to an address is not just an action of a compiler or assembler during compile time. Instead, the execution time code for VARIABLE actively places the address of the variable onto the top of the parameter stack.

The compile time action (Sequence 2) for VARIABLE is to create a definition containing the variable name, a pointer to the execution time code for variable, and the 16-bit location. One definition is created per variable name. VARIABLE is used in the form:

n VARIABLE cccc

where n is the variable's initial value and cccc is its assigned name.

Note: Although it is possible to specify at compile time (Sequence 2) the initial value of a variable, it is a much safer programming practice to always initialize variables during program initialization.

The execution time (Sequence 3) action of VARIABLE, when the variable name is referenced, is to place the address of the named 16-bit location onto the top of the parameter stack.

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains a 16-bit value.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3):

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the variable location.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) -- The dictionary has grown into the Terminal Input Buffer.

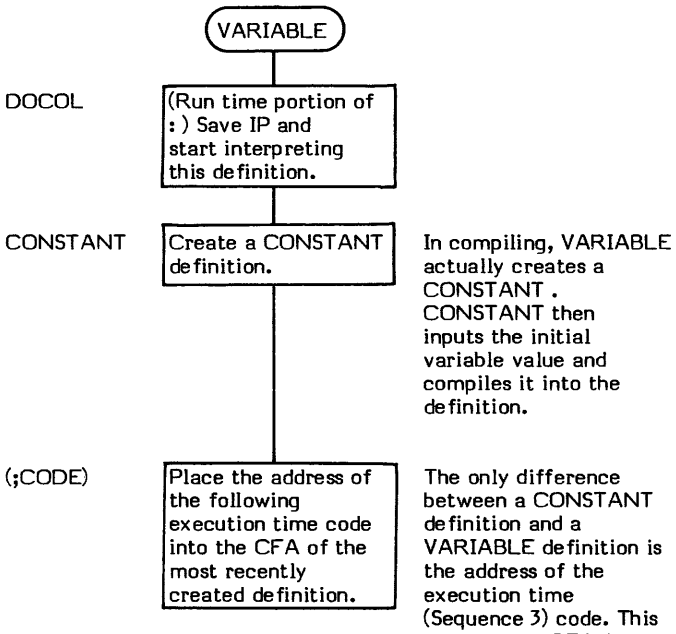
DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

VARIABLE is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for VARIABLE is VARIABLE , although in FORTH-79 no initial value may be specified at compile time.

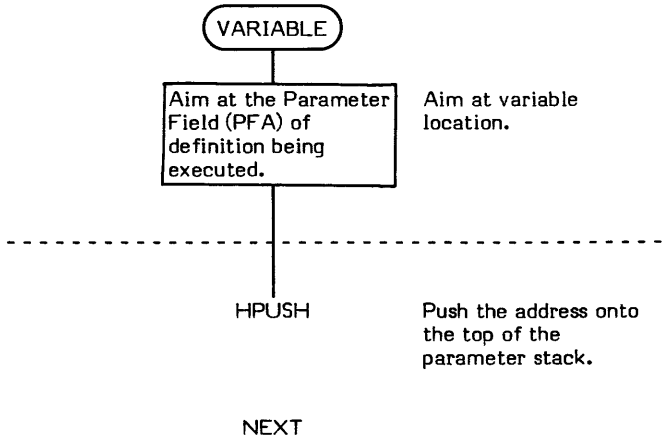
Definition:
```
: VARIABLE  ( n -- )  ( compile time )
      CONSTANT  ;CODE
```
Note: The assembly language execution time code follows the ;CODE .

COMPILE TIME action of VARIABLE : (n --)
(Sequence 2)



EXECUTION TIME action of VARIABLE : (-- address)
(Sequence 3)

NOTE: The execution time (Sequence 3) code for VARIABLE physically follows the ;CODE in the source code for VARIABLE .



VLIST (—)

VLIST (pronounced "V-list") lists the names of all of the definitions in the CONTEXT vocabulary onto the output device.

Pressing any terminal key will terminate the listing. Basically VLIST works by finding the end definition in the CONTEXT vocabulary and then chaining completely through the vocabulary via a BEGIN-UNTIL loop. Inside this loop, ID. is used to print each definition name. The output format of a VLIST generally varies between specific FORTH implementations.

* **At entry** - No parameters.
* **At exit** - No parameters.

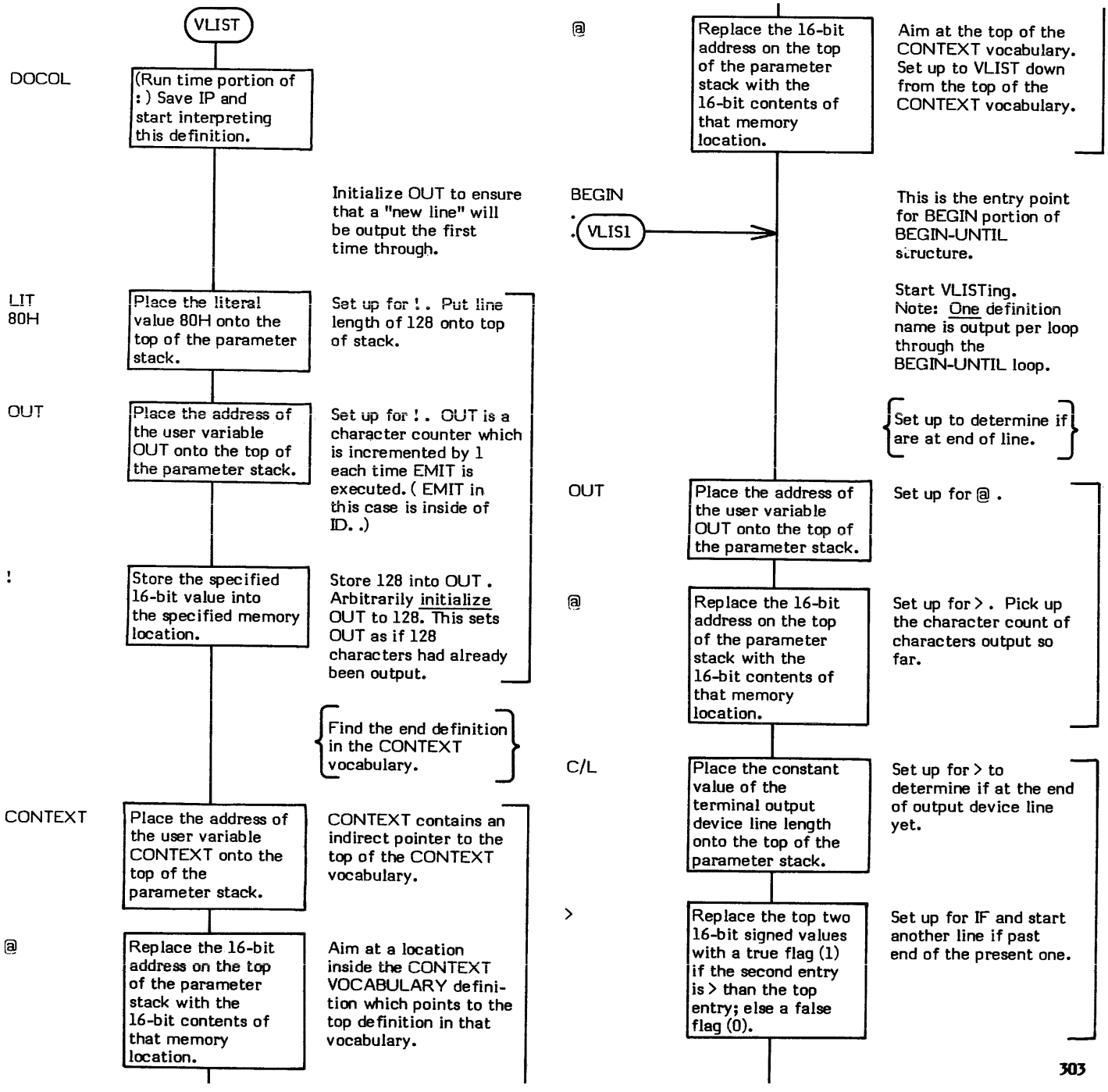
VLIST is a high level colon definition.

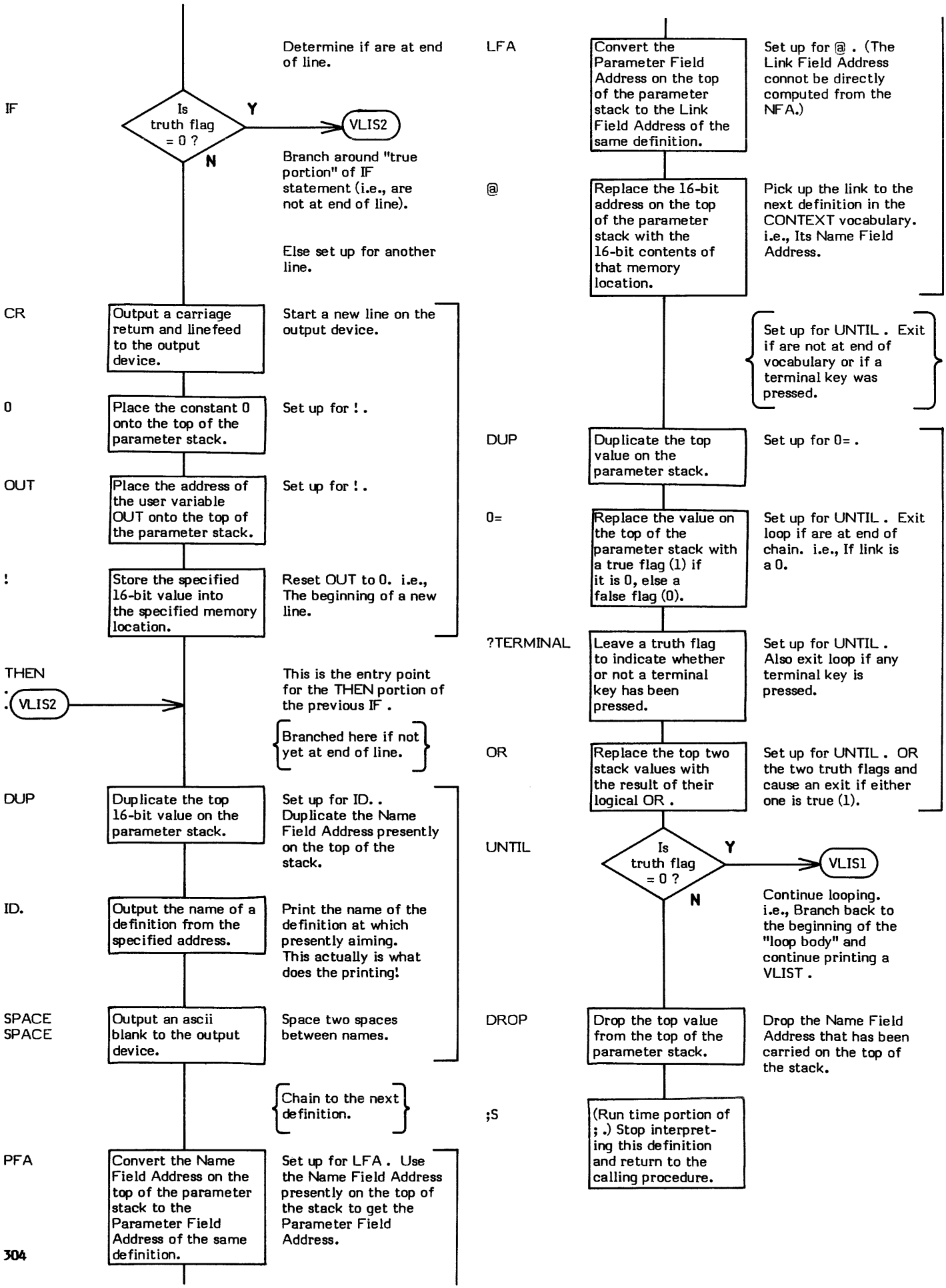
FORTH-79: VLIST is not explicitly defined by FORTH-79 but it is listed in the "FORTH-79 Referenced Word Set".

Note that although this flowchart for VLIST is long, no high level flowchart accompanies it because its logic is straightforward.

Definition:

```
:  VLIST  ( -- )
      80 OUT !   CONTEXT @ @
      BEGIN
         OUT @   C/L >   IF   CR 0 OUT !   THEN
         DUP ID.   SPACE SPACE   PFA LFA @
         DUP 0= ?TERMINAL OR
      UNTIL   DROP   ;
```




IF
Is truth flag = 0 ?
Y
N
VLIS2
Determine if are at end of line.
Branch around "true portion" of IF statement (i.e., are not at end of line).
Else set up for another line.
CR
Output a carriage return and linefeed to the output device.
Start a new line on the output device.
0
Place the constant 0 onto the top of the parameter stack.
Set up for ! .
OUT
Place the address of the user variable OUT onto the top of the parameter stack.
Set up for ! .
!
Store the specified 16-bit value into the specified memory location.
Reset OUT to 0. i.e., The beginning of a new line.
THEN
VLIS2
This is the entry point for the THEN portion of the previous IF .
Branched here if not yet at end of line.
DUP
Duplicate the top 16-bit value on the parameter stack.
Set up for ID. . Duplicate the Name Field Address presently on the top of the stack.
ID.
Output the name of a definition from the specified address.
Print the name of the definition at which presently aiming. This actually is what does the printing!
SPACE
SPACE
Output an ascii blank to the output device.
Space two spaces between names.
Chain to the next definition.
PFA
Convert the Name Field Address on the top of the parameter stack to the Parameter Field Address of the same definition.
Set up for LFA . Use the Name Field Address presently on the top of the stack to get the Parameter Field Address.
LFA
Convert the Parameter Field Address on the top of the parameter stack to the Link Field Address of the same definition.
Set up for @ . (The Link Field Address connot be directly computed from the NFA.)
@
Replace the 16-bit address on the top of the parameter stack with the 16-bit contents of that memory location.
Pick up the link to the next definition in the CONTEXT vocabulary. i.e., Its Name Field Address.
Set up for UNTIL . Exit if are not at end of vocabulary or if a terminal key was pressed.
DUP
Duplicate the top value on the parameter stack.
Set up for 0= .
0=
Replace the value on the top of the parameter stack with a true flag (1) if it is 0, else a false flag (0).
Set up for UNTIL . Exit loop if are at end of chain. i.e., If link is a 0.
?TERMINAL
Leave a truth flag to indicate whether or not a terminal key has been pressed.
Set up for UNTIL . Also exit loop if any terminal key is pressed.
OR
Replace the top two stack values with the result of their logical OR .
Set up for UNTIL . OR the two truth flags and cause an exit if either one is true (1).
UNTIL
Is truth flag = 0 ?
Y
N
VLIS1
Continue looping. i.e., Branch back to the beginning of the "loop body" and continue printing a VLIST .
DROP
Drop the top value from the top of the parameter stack.
Drop the Name Field Address that has been carried on the top of the stack.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

VOC-LINK (— data address)

VOC-LINK (pronounced "voke-link" for vocabulary link) is a user variable that contains an address that points to a field in the vocabulary definition of the last vocabulary created. Do not confuse VOC-LINK with the vocabulary pseudo link field. This voc-link field in the vocabulary definition in turn points to the voc-link field in the next previously created vocabulary definition. This is to say that all vocabulary definitions are linked together in chronological order via this pointer. The FORTH vocabulary is always the end of the voc-link chain.

VOC-LINK is described in more detail in the description of VOCABULARY .

VOC-LINK is initialized by COLD during system startup with data from the origin parameter area.

The user variable VOC-LINK is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable VOC-LINK .

Refer to VOCABULARY , FORTH , and USER .

FORTH-79: There is no FORTH-79 equivalent for VOC-LINK .

VOCABULARY

VOCABULARY

DEFINITION TIME (Sequence 1): (—)

COMPILE TIME (Sequence 2): (—)

EXECUTION TIME (Sequence 3): (—)

VOCABULARY is a defining word and therefore exhibits three different sets of actions; those actions at definition time, those actions at compile time, and those at execution time.

The definition time action of VOCABULARY is to create the word VOCABULARY (a "parent" definition).

The compile time action of VOCABULARY is to create a vocabulary definition ("child" definitions). These definitions then serves as the head or first entry of a new vocabulary.

The execution time action of VOCABULARY is to store the address of that vocabulary's "Vocabulary Link Pointer" into the user variable CONTEXT . The effect of this is to cause the named vocabulary to be searched first whenever a dictionary search is performed. (Refer to -FIND .)

The purpose of VOCABULARY is best understood by first understanding the use of vocabularies by FORTH in general.

A "vocabulary" is the mechanism used by FORTH to limit the scope of a name by restricting dictionary searches to a specified subset of the dictionary. This ability to make names non-global is common among high level languages as can be seen in the "scoping" rules in ALGOL, PASCAL, and PL/1, where a routine's internal variables are inaccessible to higher-level routines. In FORTH, however the programmer must explicitly "hide" words using vocabulary names in conjunction with the word DEFINITIONS .

Each word in the dictionary is actually identified by two names:

1. The word's vocabulary name/branch name (analogous to its surname or family name).

2. The word's definition name (analogous to its given name).

If a definition name is unique throughout the entire dictionary, only the definition name is necessary to identify the word. However, if there is more than one definition with the same name, the vocabulary name must also be used.

Vocabularies in FORTH can be best envisioned as two-dimensional branches of a tree structure with the main trunk being the FORTH vocabulary. (Refer to Figure VOC-1.) Any single vocabulary branch may be the "parent" to any number of "child" branches. A new definition may be appended at any time to the end of any vocabulary.

These branches are what make scoping possible. Vocabularies cause the one-dimensional dictionary to be searched in a logically two-dimensional order. This vocabulary "tree" is searched inwardly (traversed) in one direction from the last definition of the specified vocabulary inward toward the main trunk of the tree (the FORTH vocabulary). The search proceeds in a direct line from "child" to "parent" to "grandparent". Vocabularies not in this direct lineage are not searched; therefore, names contained within them are not within the scope of the search. (Refer to Figure VOC-1.)

The role of the VOCABULARY definition starts to become apparent when one attempts to implement this two-dimensional tree structure into the FORTH dictionary. The dictionary is a one-dimensional stack, allocated with one dictionary pointer (DP). Therefore, this two-dimensional tree structure must be logically built and maintained within a one-dimensional environment (the dictionary).

The logical vocabulary tree structure must support several functions:

1. A mechanism must exist to allow a "child" vocabulary to be created and appended to its "parent" vocabulary.

2. It must be possible to logically append a new definition to the end of any vocabulary while actually physically appending the definition to the end of the dictionary "stack".

3. Vocabularies must be chained so that a dictionary search proceeds inwardly through the "child" to the "parent" vocabularies.

4. It must be possible to specify (i.e., name) both the vocabulary to which a new definition is to be appended and a possibly separate vocabulary which contains the words to be used in creating the new definition.

All of this is made possible through the use of the "Vocabulary Link Field" pointer contained in the VOCABULARY definition. (Refer to Figure VOC-2.) This pointer serves exactly the same purpose as the Link Field in other definitions. Note, however, that this Vocabulary Link Field is contained in the Parameter Field of the VOCABULARY definition. It is not to be confused with the normal Link Field contained in all definition headers.

When the VOCABULARY definition is created (refer to Figure VOC-3), this pointer is initially linked to the Pseudo Name Field in its parent"s VOCABULARY definition. (Remember that Link Fields always point to Name Fields.) This satisfies the requirement that a "child" be logically linked to its "parent".

When a definition is first appended to this newly created vocabulary (refer to Figure VOC-4), CREATE copies the pointer from the Vocabulary Link Field into the new definition's normal header Link Field. The Vocabulary Link Field is then set to point to the Name Field of the new definition thereby linking the new definition into the vocabulary chain. As new words are physically added to the end of the dictionary; they, too, are logically linked into the end of this (the "current" vocabulary) chain and the Vocabulary Link Field is re-linked to them. In other words, as new definitions are appended to the vocabulary, the vocabulary link pointer is altered to always point to the last (latest) word in the vocabulary. This satisfies the requirement that definitions be physically added to the

one-dimensional dictionary while being logically linked in a two-dimensional tree structure. The link between "child" and "parent" is still preserved. Note that the normal header linkages are used to link the newly created vocabulary definition into the one-dimensional dictionary "stack".

Another requirement is that searches must proceed inwardly through the "child" to the "parent" vocabularies. When searching, -FIND first examines the Name Field of a definition for a character string match. If there is no match, that definition's Link Field is used to obtain the Name Field Address of the next definition "up" the chain where the comparison is performed again. The vocabulary definition is structured such that it takes advantage of -FIND 's characteristic of continuing the search if a name match is not found. The "Pseudo Name Field" is actually a perfectly legal and valid but impossible name. (Refer to Figure VOC-5.) This Name Field is followed by a valid "Link Field" which points to the last definition in that vocabulary. If a "child" vocabulary is searched for a word which it does not contain, the search will continue until the topmost definition is reached. The Link Field of this definition, as previously stated, points to the Pseudo Name Field of the "parent" vocabulary. Since there is no match, -FIND then examines the Pseudo Name Field of the "parent" vocabulary definition. Since this Name Field contains an ascii blank, which by definition is an impossible name; no match is found and the contents of the following Vocabulary Link Field is then fetched. -FIND has been "tricked" and never "realizes" that the Pseudo Name Field and the Vocabulary Link Fields are not common header Name Fields and Link Fields. Since the Vocabulary Link Field always points to the latest definition in the vocabulary, the search is continued at the "bottom" of the "parent" vocabulary. This means that the search has proceeded inwardly from the "child" to the "parent". This chaining through vocabularies will continue until a character match or null Link Field is encountered.

The Vocabulary Link Field is also the means by which it is possible to specify to which vocabulary a definition is to be appended and which vocabulary is to be searched first when creating a definition. To do this, two additional pointers are used: the user variables CURRENT and CONTEXT. These two variables contain pointers to Vocabulary Link Fields.

The CURRENT pointer is used by CREATE and is set to point to the vocabulary into which definitions are "currently" being appended. The CONTEXT pointer is set to point to the vocabulary which should be searched first. (-FIND first picks up the contents of CONTEXT which is aiming at a Vocabulary Link Field that in turn is always aiming at the "latest" word of that vocabulary. If -FIND has not found a match by the time it encounters a null link, usually the end of the FORTH vocabulary; then the address in the CURRENT pointer is fetched and the search is repeated starting at the bottom of the CURRENT vocabulary.)

CONTEXT is set by naming a previously defined vocabulary thereby executing the run time portion of VOCABULARY.

The execution time action (Sequence 3) of VOCABULARY is to store the address of that vocabulary's "Vocabulary Link Pointer" into the user variable CONTEXT . This causes the named vocabulary to be searched first whenever a dictionary search is performed.

This makes for very readable FORTH programs because simply stating the name of a vocabulary "automatically" makes it the vocabulary to be searched first.

It should be noted that CONTEXT simply denotes which vocabulary is to be searched first (i.e., sets the search scope). This means that its use is not limited to only specifying which vocabulary is search first when creating new definitions. It can also be used to specify which definitions are to be executed. For example, the EDITOR vocabulary contains words used for editing. Stating EDITOR causes the EDITOR vocabulary to be searched, thereby allowing editing commands to be found and executed.

CURRENT is set via the word DEFINITIONS . DEFINITIONS copies the contents of CONTEXT into CURRENT . This then is what determines onto which vocabulary "definitions" are to be appended. The compile time action of : is to copy the contents of CURRENT into CONTEXT . This makes the vocabulary to which definitions are being apprended also the first vocabulary to be searched. There is no special reason why colon does this except that it is common practice to append to and search from the same vocabulary.

For example: VOCABULARY TOAD creates a linked vocabulary definition named TOAD . Later execution of TOAD causes CONTEXT to be set so that dictionary searching will begin with TOAD . Execution of the word DEFINITIONS would then copy CONTEXT into CURRENT . i.e., TOAD will have definitions appended to it as well as being searched first. Execution of : when compiling a definition causes CURRENT to be set to CONTEXT. Naming another vocabulary within the definition, such as FROG, would cause FROG to be searched first but definitions would still be appended to TOAD (because of the previous DEFINITIONS).

One additional "pointer", a definition named LATEST , also makes use of the fact that the Vocabulary Link Field always points to the last, i.e., "latest", definition added to a vocabulary. (See LATEST .) LATEST is used to obtain the address of the last definition compiled. It does this by performing an indirect fetch through CURRENT .

Note that CURRENT and CONTEXT are indirect pointers (i.e., to obtain a definition address, two fetches must be performed). The contents of the variable must be fetched to point to the Vocabulary Link Field. Then secondly, the contents of the Link Field must be fetched to point to the definition itself.

As shown in Figure VOC-1, there is one other pointer contained within the vocabulary definition: the "Chronological Link Pointer". When a vocabulary definition is compiled (Sequence 2), the "Chronological Link Pointer" is aimed at its corresponding location in the previously created vocabulary definition. This causes all vocabularies to be chronologically linked together in the order they were created. The "Chronological Link Pointer" of the definition being created is filled with the contents of the user variable VOC-LINK . VOC-LINK is then stuffed with the address of the "chronological link pointer" of the definition being created where it can then be used to link the next vocabulary definition into the chain.

Some versions of fig-FORTH use this pointer to allow forgetting through multiple vocabularies. WARNING! When multiple vocabularies have been created, the use of the normal fig-Model FORGET produces indeterminable results and will probably cause an eventual system crash. Refer to FORGET .

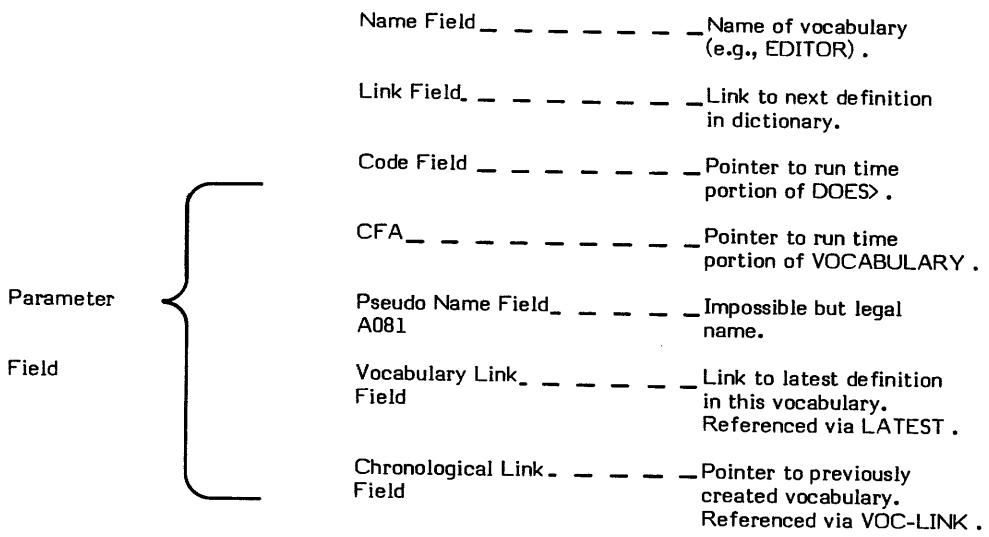


Figure VOC-1

Structure of a VOCABULARY Definition

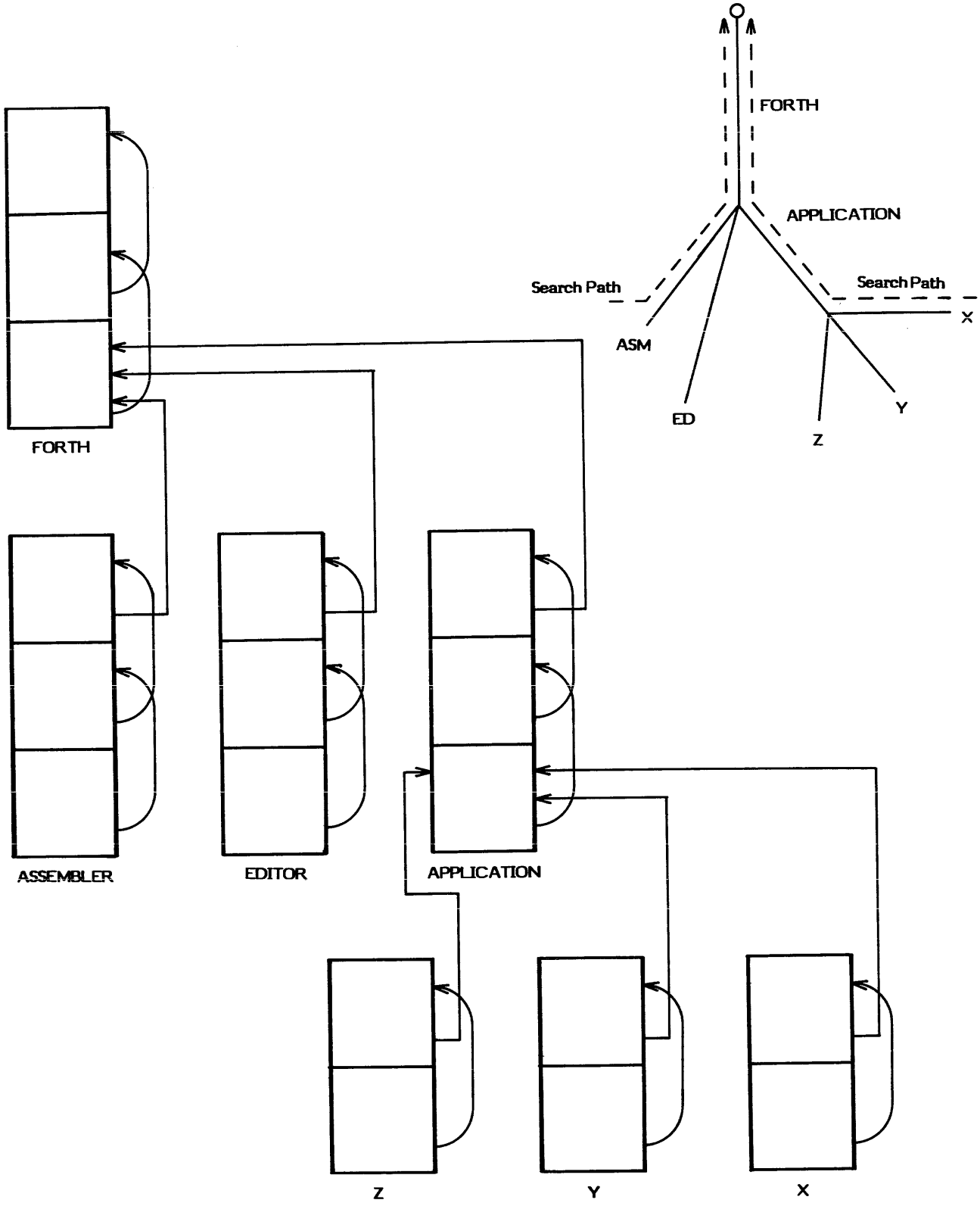


Figure VOC-2

Logical and Physical Vocabulary Linkage
(Reprinted by permission of Kim Harris)

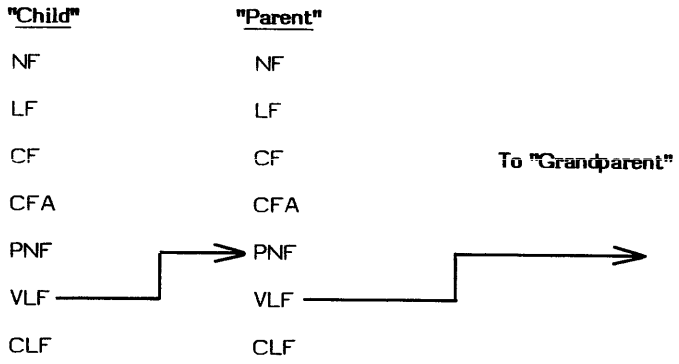
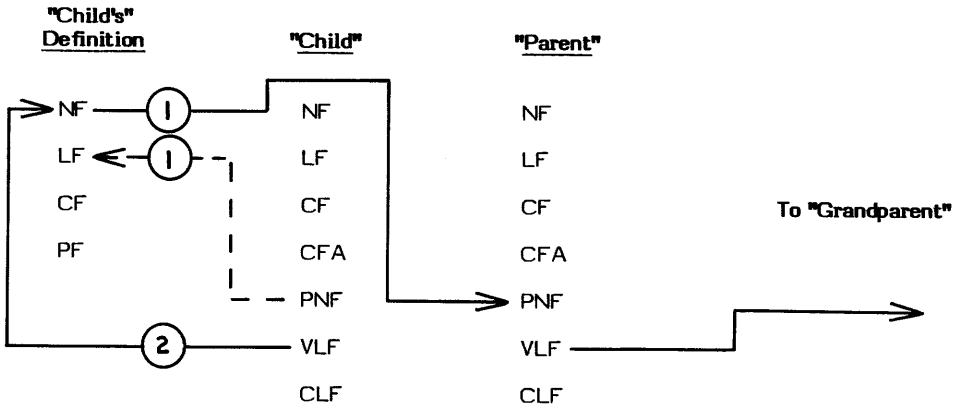


Figure VOC-3

The newly created "child" vocabulary definition is linked to its "parent" vocabulary definition.

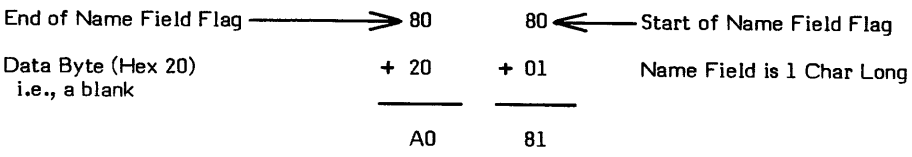


1. The "child's" Vocabulary Link Field is copied into the Link Field of the new definition. This links the "child's" definition to the "parent's".

2. The "child's" Vocabulary Link Field is then aimed at the new definition's Name Field. This links the "child" to its definitions.

Figure VOC-4

Adding a Definition to a Vocabulary.



This is an "impossible" Name Field since the Interpreter ignores blanks, therefore a blank cannot normally be the name of a definition.

NOTE: Bytes are swapped in the 8080 when stored in memory. A081 will be 810A when in memory.

Figure VOC-5

Structure of the Pseudo Name Field

DEFINITION TIME (Sequence 1) - When VOCABULARY is created:

* **At entry** - No parameters.
* **At exit** - No parameters.

COMPILE TIME (Sequence 2) - When VOCABULARY is used to create a vocabulary definition:

* **At entry** - No parameters on the parameter stack; however, the word VOCABULARY must be followed by a character string specifying the name of the vocabulary to be created.
* **At exit** - No parameters.

EXECUTION TIME (Sequence 3) - When the definition created via VOCABULARY executes:

* **At entry** - No parameters.
* **At exit** - No parameters on the parameter stack; however, the user variable CONTEXT will be pointing to the Vocabulary Link Field in the named vocabulary's definition.

LIKELY ERROR MESSAGES:

DICTIONARY FULL (2) -- The dictionary has grown into the Terminal Input Buffer.

DEFINITION NOT FINISHED (14H) -- The position of the parameter stack pointer is not the same as it was when this definition started being compiled. Something is wrong with the definition.

VOCABULARY is a high level colon definition.

Refer to CONTEXT , CURRENT , DEFINITIONS , LATEST , FORTH , VOC-LINK . -FIND , and FORGET .

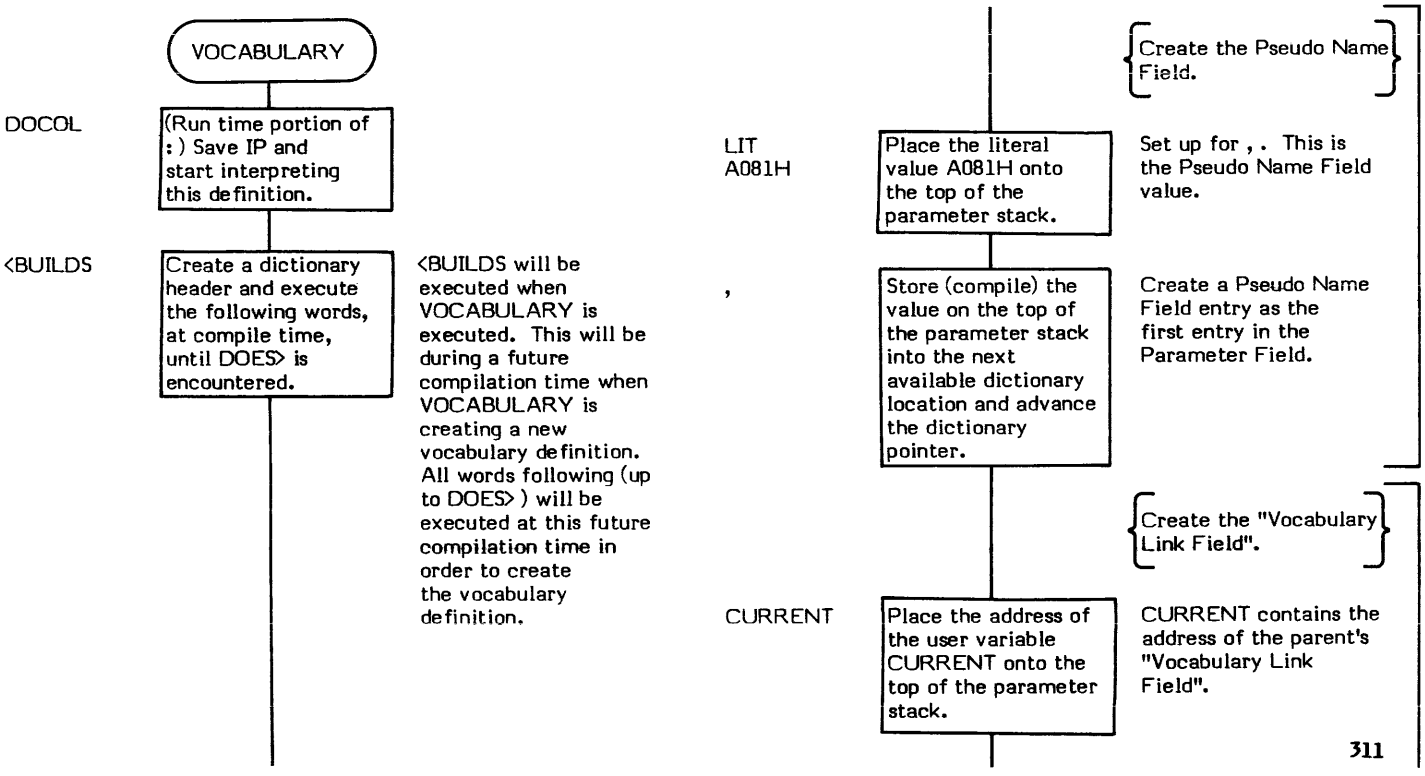
FORTH-79: The FORTH-79 equivalent for VOCABULARY is VOCABULARY . Note, however, that the structure of vocabularies differs from the fig-FORTH Model. Refer to the FORTH-79 Standard.

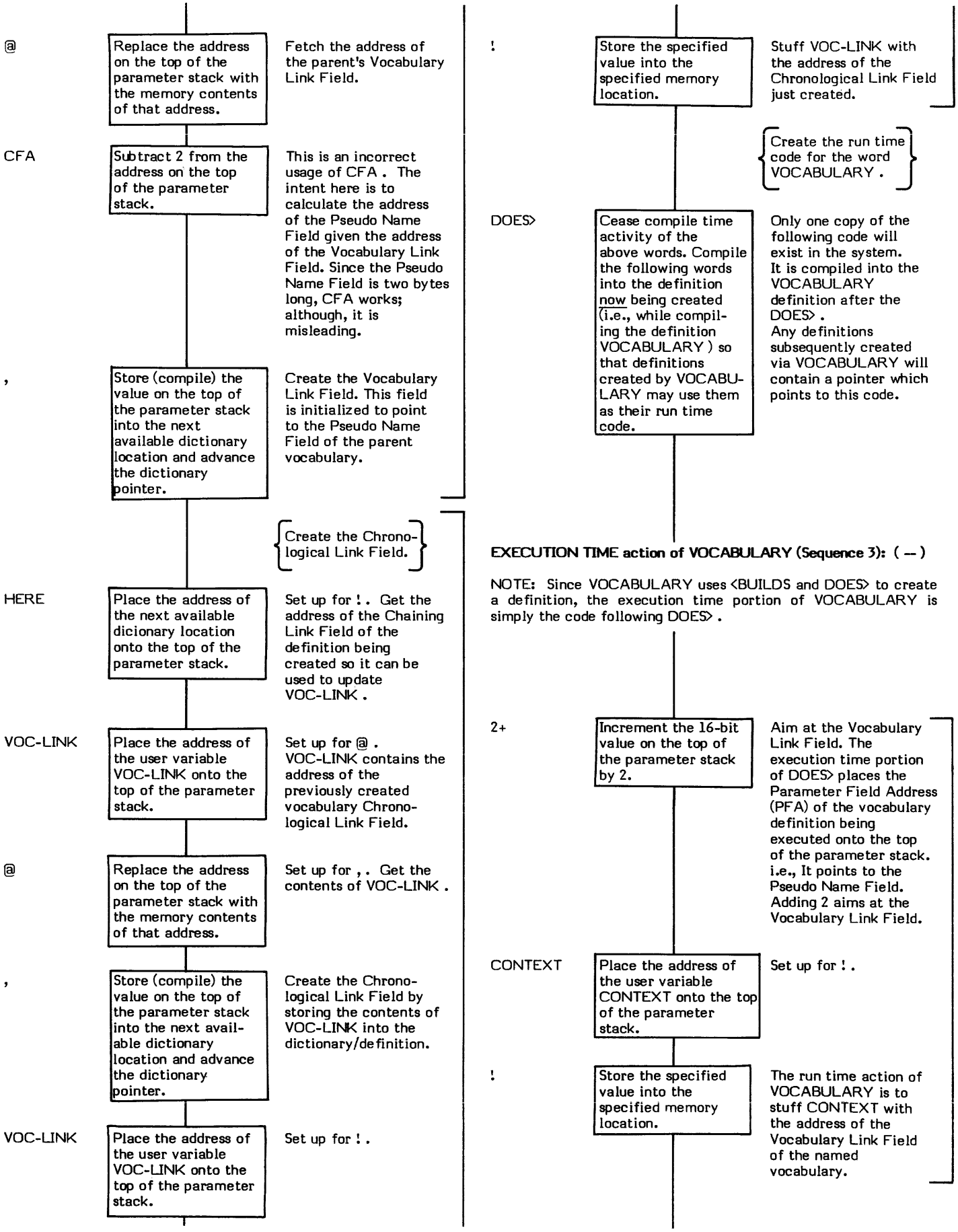
Definition:

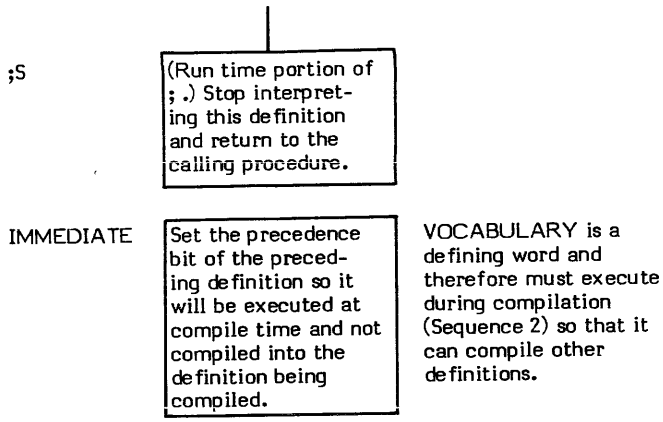
```
:  VOCABULARY  ( -- )
       <BUILDS
           A081 ,   CURRENT @ CFA ,
           HERE VOC-LINK @ , VOC-LINK !
       DOES>
           2+ CONTEXT !   ;   IMMEDIATE
```

DEFINITION/COMPILE TIME action of VOCABULARY (Sequence 1/2): (--)

NOTE: The act of compiling all of the words in VOCABULARY 's flowchart is the definition time (Sequence 1) activity of VOCABULARY . The words between <BUILDS and DOES> are the compile time (Sequence . 2) activity of VOCABULARY . The words between DOES> and ;S are the execution time (Sequence 3) action of VOCABULARY.




@
Replace the address on the top of the parameter stack with the memory contents of that address.
Fetch the address of the parent's Vocabulary Link Field.
CFA
Subtract 2 from the address on the top of the parameter stack.
This is an incorrect usage of CFA . The intent here is to calculate the address of the Pseudo Name Field given the address of the Vocabulary Link Field. Since the Pseudo Name Field is two bytes long, CFA works; although, it is misleading.
,
Store (compile) the value on the top of the parameter stack into the next available dictionary location and advance the dictionary pointer.
Create the Vocabulary Link Field. This field is initialized to point to the Pseudo Name Field of the parent vocabulary.
Create the Chrono-logical Link Field.
HERE
Place the address of the next available dicionary location onto the top of the parameter stack.
Set up for ! . Get the address of the Chaining Link Field of the definition being created so it can be used to update VOC-LINK .
VOC-LINK
Place the address of the user variable VOC-LINK onto the top of the parameter stack.
Set up for @ . VOC-LINK contains the address of the previously created vocabulary Chrono-logical Link Field.
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Set up for , . Get the contents of VOC-LINK .
,
Store (compile) the value on the top of the parameter stack into the next avail-able dictionary location and advance the dictionary pointer.
Create the Chrono-logical Link Field by storing the contents of VOC-LINK into the dictionary/definition.
VOC-LINK
Place the address of the user variable VOC-LINK onto the top of the parameter stack.
Set up for ! .
!
Store the specified value into the specified memory location.
Stuff VOC-LINK with the address of the Chronological Link Field just created.
Create the run time code for the word VOCABULARY .
DOES>
Cease compile time activity of the above words. Compile the following words into the definition now being created (i.e., while compil-ing the definition VOCABULARY) so that definitions created by VOCABU-LARY may use them as their run time code.
Only one copy of the following code will exist in the system. It is compiled into the VOCABULARY definition after the DOES> . Any definitions subsequently created via VOCABULARY will contain a pointer which points to this code.
EXECUTION TIME action of VOCABULARY (Sequence 3): (--)
NOTE: Since VOCABULARY uses <BUILDS and DOES> to create a definition, the execution time portion of VOCABULARY is simply the code following DOES> .
2+
Increment the 16-bit value on the top of the parameter stack by 2.
Aim at the Vocabulary Link Field. The execution time portion of DOES> places the Parameter Field Address (PFA) of the vocabulary definition being executed onto the top of the parameter stack. i.e., It points to the Pseudo Name Field. Adding 2 aims at the Vocabulary Link Field.
CONTEXT
Place the address of the user variable CONTEXT onto the top of the parameter stack.
Set up for ! .
!
Store the specified value into the specified memory location.
The run time action of VOCABULARY is to stuff CONTEXT with the address of the Vocabulary Link Field of the named vocabulary.


;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of the preceding definition so it will be executed at compile time and not compiled into the definition being compiled.
VOCABULARY is a defining word and therefore must execute during compilation (Sequence 2) so that it can compile other definitions.

W

W

W is a pointer used by FORTH's "threading" words. It plays an important role in such words as NEXT , ;S , DOCOL (the execution time portion of :), etc.

W generally serves as a pointer to the Code Field of the definition currently being executed. NEXT jumps indirectly "through" this pointer to execute the Code Field procedure.

W is not a true FORTH word. It is not a variable. It is a logical entity and may be physically kept in registers or memory or whatever depending upon the exact system implementation.

In 8080 fig-FORTH Version 1.1, W is contained in register pair DE.

Refer to NEXT , : , and ;S .

WARNING (— data address)

WARNING is a user variable that contains a value used in determining the following:

1. Whether the system ABORT 's or QUIT 's via the word ERROR .
2. The format of messages output via the word MESSAGE .

When ERROR executes and WARNING contains a negative value, (ABORT) will occur and execution will stop. (ABORT) is intended to be a user defined error handling procedure.

When ERROR executes and WARNING contains a non-negative value (i.e., zero or positive value), MESSAGE will execute and then a QUIT will stop execution.

When MESSAGE executes and WARNING contains a zero, only the message number will be output. (This allows the system to operate without a disk.)

When MESSAGE executes and WARNING is non-zero, text messages will be output using Line 0 of Screen 4 of Drive 0 as a base location.

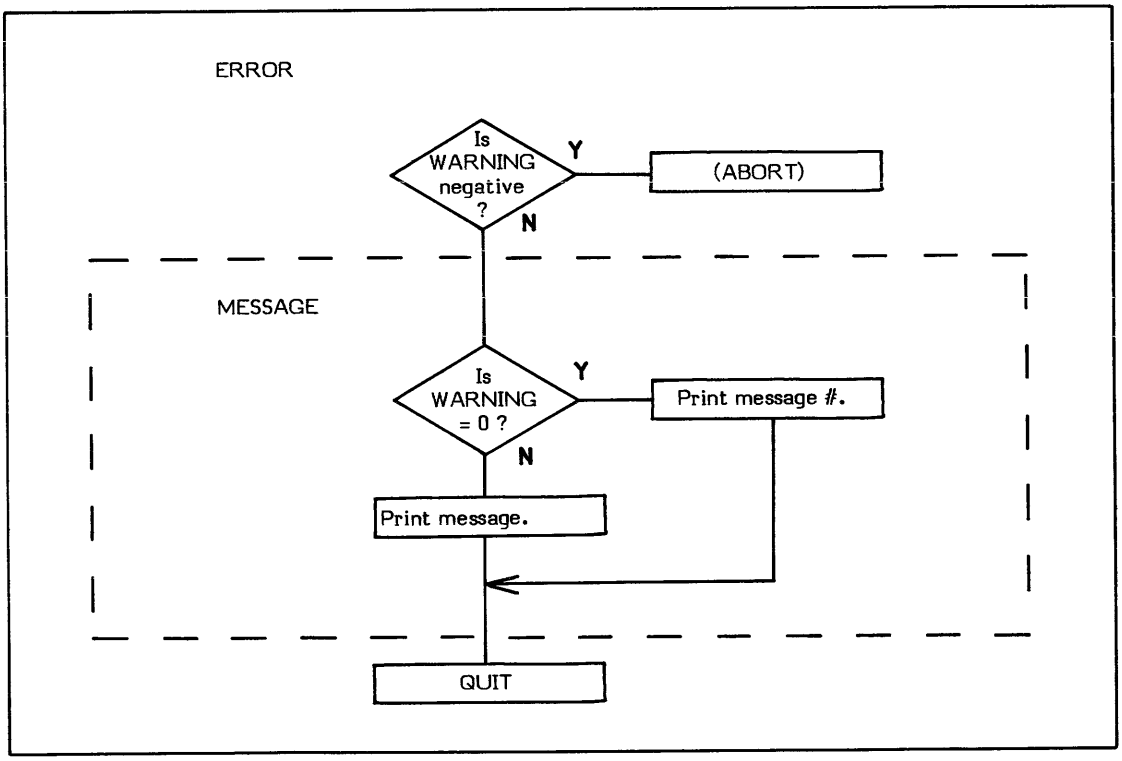
WARNING is initialized by COLD during system startup with data from the origin parameter area.

The user variable WARNING is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable WARNING .

Refer to MESSAGE , ERROR , and USER .

FORTH-79: There is no FORTH-79 equivalent for WARNING .



How WARNING is Used

WHILE

WHILE

COMPILE TIME (Sequence 2): (loop address\ 1 -- loop address\ 1\ offset location\ 4)

EXECUTION TIME (Sequence 3): (truth flag --)

WHILE is a compiler word and therefore exhibits two different sets of actions; those actions at compile time and those at execution time.

WHILE is used as the beginning of the "true portion/loop body" of a BEGIN-WHILE-REPEAT structure in the form:

```
BEGIN     "set exit conditional"
WHILE     "true portion/loop body"
REPEAT
```

This is known as a "pre-test" loop.

It is important to note that when using this structure it is possible for the loop to terminate before the "loop body" is executed even once. This is because the exit condition is tested before executing the "loop body". i.e., The loop will continue "while" the boolean flag is true.

If the "loop body" must always be executed at least once (a post-test loop), the structure BEGIN-UNTIL should be used instead.

The compile time (Sequence 2) action of WHILE is almost identical to that of IF . So close, in fact, that an IF is executed within WHILE at compile time. (This IF is executed when compiling a definition containing WHILE , at Sequence 2; not during compilation of WHILE , which would have been at Sequence 1.) A 0BRANCH is compiled into the definition and the location immediately following is reserved for the 0BRANCH 's forward branch offset. The address of that location is placed on the stack so that the following REPEAT can calculate and store the appropriate branch offset into the reserved location.

To provide compiler security, the value 4 is left on the stack. (The 2 left by IF is incremented by 2.) REPEAT can then check for the presence of this value 4. This provides a somewhat secure (but not foolproof) method of checking for an unmatched WHILE and REPEAT .

The execution time (Sequence 3) action of WHILE is to control exit from the loop. This is accomplished via the compiled 0BRANCH . A boolean flag is its input and action is taken based on this flag. If the flag is true (non-zero), the true portion of the structure is executed (i.e., just as in IF). If the flag is false (0), the loop will be exited and control will be passed to the word immediately following REPEAT .

BEGIN-WHILE-REPEAT structures must be used within a colon definition.

Any amount of processing may be performed between the BEGIN and the WHILE as long as the final result is a boolean flag.

Note that WHILE is an IMMEDIATE word. This means that its precedence bit is set, and it will therefore execute at compile time.

(NUMBER) is an example of a word which uses WHILE .

COMPILE TIME (Sequence 2):

* **At entry** - The top of the parameter stack contains the 16-bit signed single precision value 1 used for compiler security. The second stack entry contains the 16-bit entry point address of the first word following BEGIN . These parameters are provided by BEGIN .

* **At exit** - The top of the parameter stack contains the 16-bit signed single precision value 4. This value is used by REPEAT for compiler security to ensure that the BEGIN-WHILE-REPEAT structure contains a WHILE . The second stack entry contains the 16-bit address of the reserved 0BRANCH offset location created by the IF inside of WHILE . The third stack entry contains the 16-bit signed single precision value 1 left by the BEGIN to ensure that the structure is started with a BEGIN . The fourth stack entry contains the 16-bit address of the first location following the BEGIN statement (i.e., the loop address). These parameters are used by REPEAT .

EXECUTION TIME (Sequence 3):

* **At entry** - The top of the parameter stack contains a 16-bit signed boolean truth flag.

* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

WHILE is a high level colon definition.

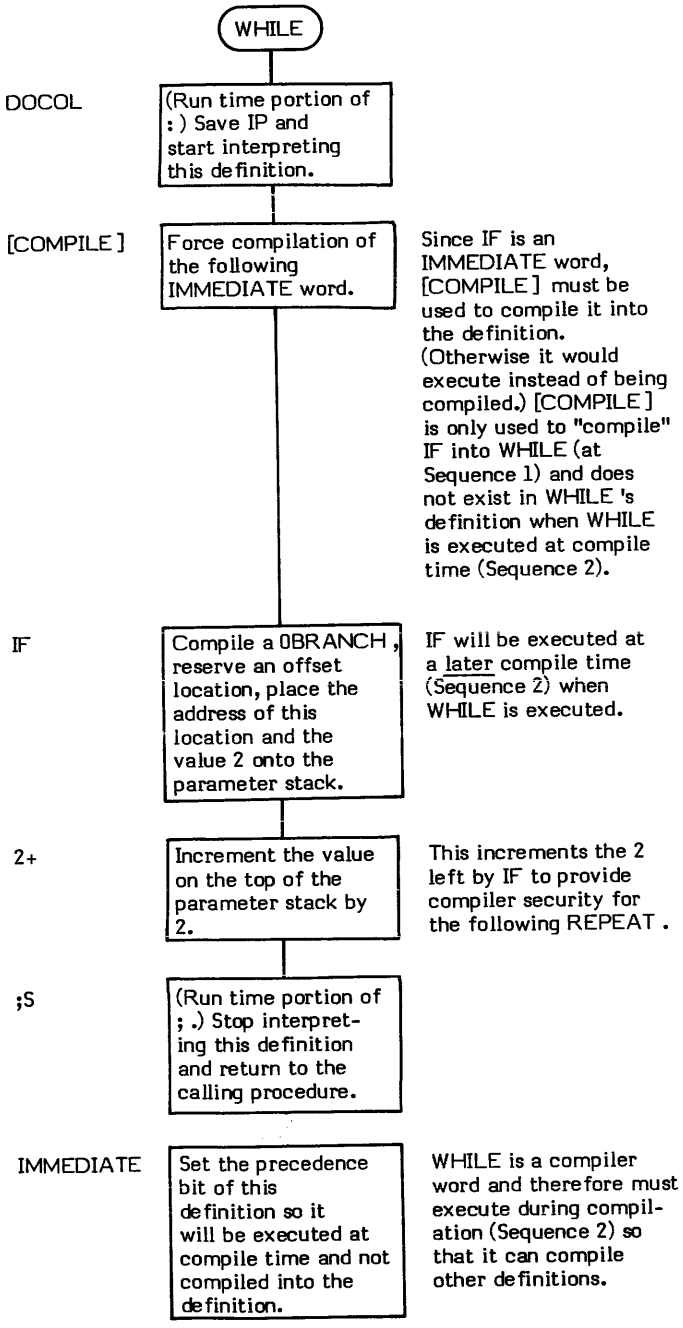
Refer to BEGIN , REPEAT , 0BRANCH , and UNTIL .

FORTH-79: The FORTH-79 equivalent for WHILE is WHILE .

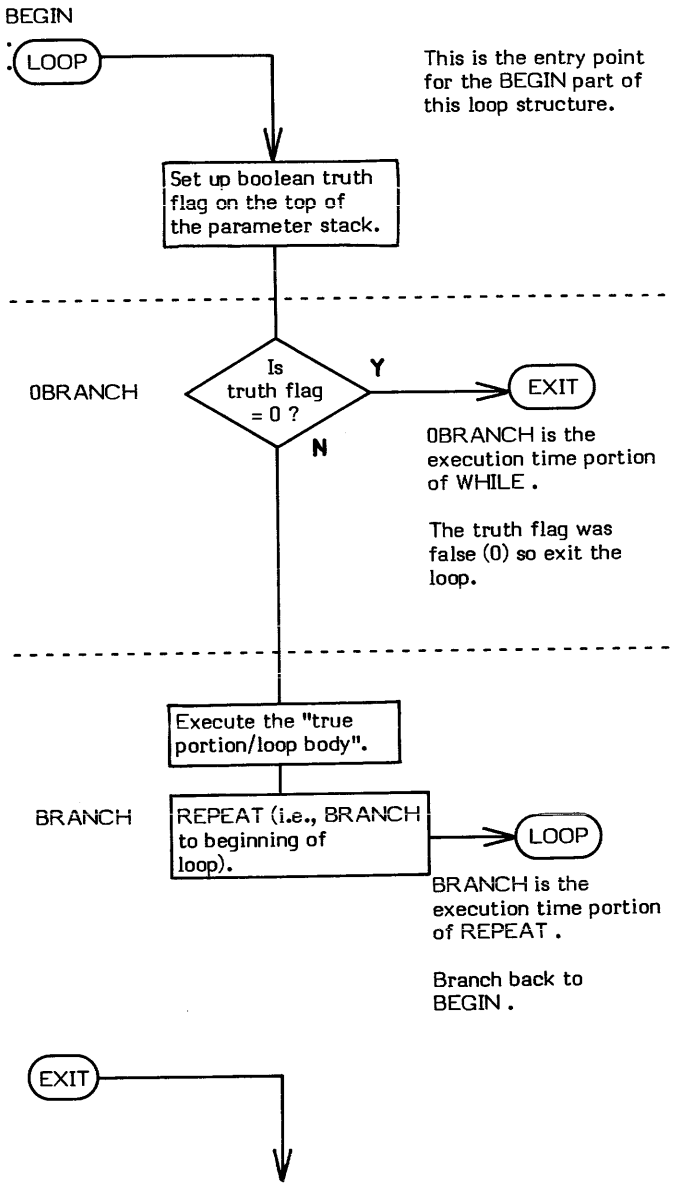
Definition:

```
:  WHILE  ( loop address\ 1 -- loop address\ 1\ offset\ 4 )   ( compile time )
       [COMPILE]  IF  2+  ;  IMMEDIATE
```

COMPILE TIME action of WHILE (Sequence 2):
(loop address \ 1 — loop address \ 1 \ offset location \ 4)



EXECUTION TIME action of WHILE (Sequence 3):
(truth flag —)



WIDTH

WIDTH (— data address)

WIDTH is a user variable that contains the maximum number of characters saved in the compilation of a definition name (i.e., the "width" of the Name Field). This value may range from 1 to 31 characters (31 is the default value). CREATE references WIDTH to determine how many characters to compile into the Name Field. The length byte is not included in the field width specified by WIDTH .

WIDTH is initialized by COLD during system startup with data from the origin parameter area.

The user variable WIDTH is stored as a 16-bit single precision value. When in memory, the high and low order bytes may be switched depending upon the processor used.

* **At entry** - No parameters.
* **At exit** - The top of the parameter stack contains the address of the user variable WIDTH .

Refer to CREATE , and USER .

FORTH-79: There is no FORTH-79 equivalent for WIDTH .

WORD (delimiter value --)

WORD parses one word from the input stream. For example, WORD reads text characters from the input stream until the specified delimiter character is encountered.

Text characters are transferred from the input stream to memory starting at the next available dictionary location (HERE).

WORD leaves the character count of the text string in the first byte of the string. The string is terminated with one or more blanks.

Leading delimiter characters are ignored and not transferred to memory. Trailing delimiters are not transferred to memory.

Ascii "nulls" (00H) are treated as unconditional delimiters. Refer to ENCLOSE for specific conditions and their results.

The source of the input stream is determined by the value of the user variable BLK . If BLK is zero, text is input from the terminal buffer. If BLK is non-zero, the value specifies the disk block to be used as input.

* **At entry** - The top of the parameter stack contains a 16-bit value. The low order 8-bits contains an 8-bit ascii character to be used as a delimiter. The high order 8-bits should be zero. BLK contains a value used in determining input stream source.

* **At exit** - No parameters on the stack. However, HERE points to the beginning of the parsed word.

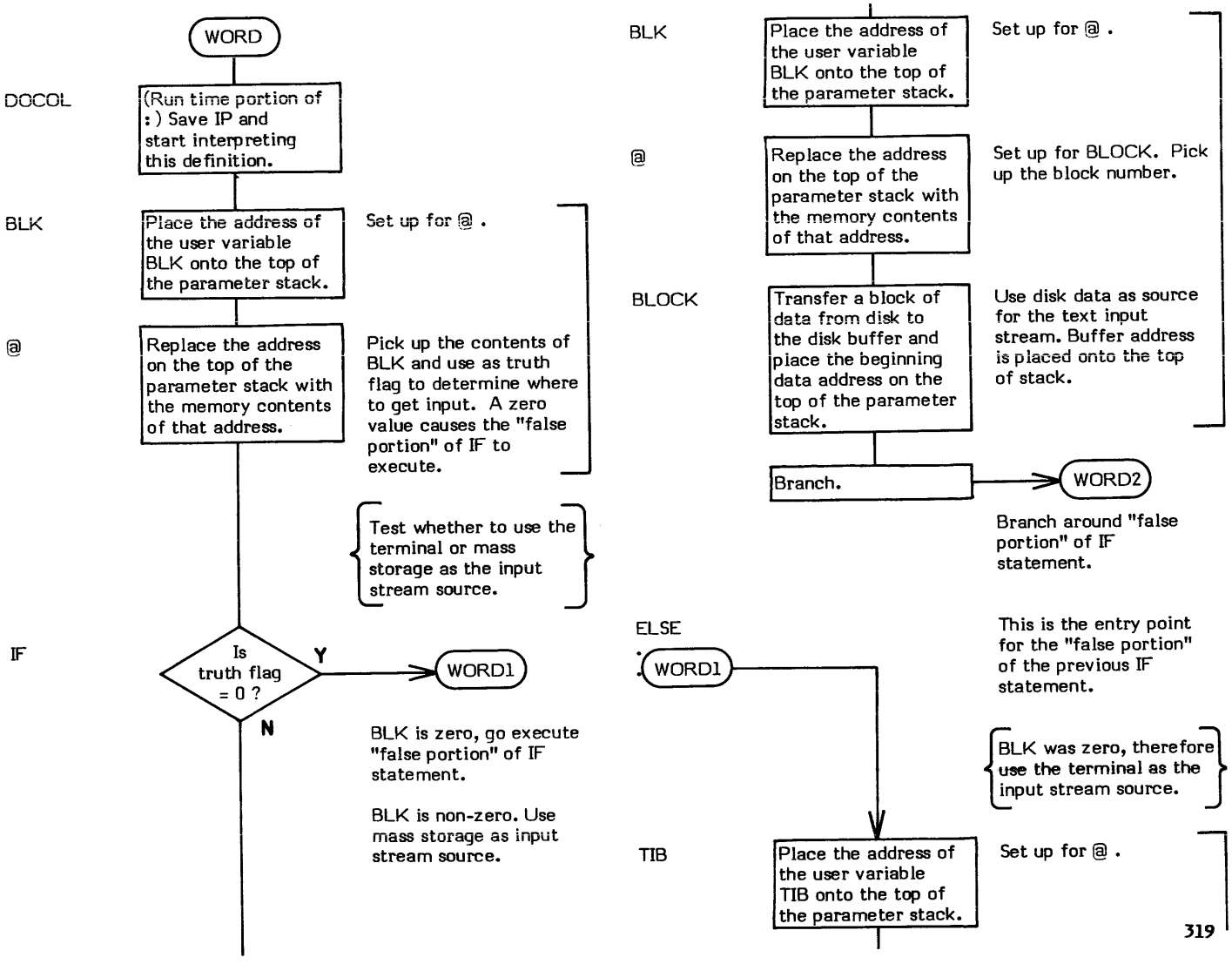
WORD is a high level colon definition.

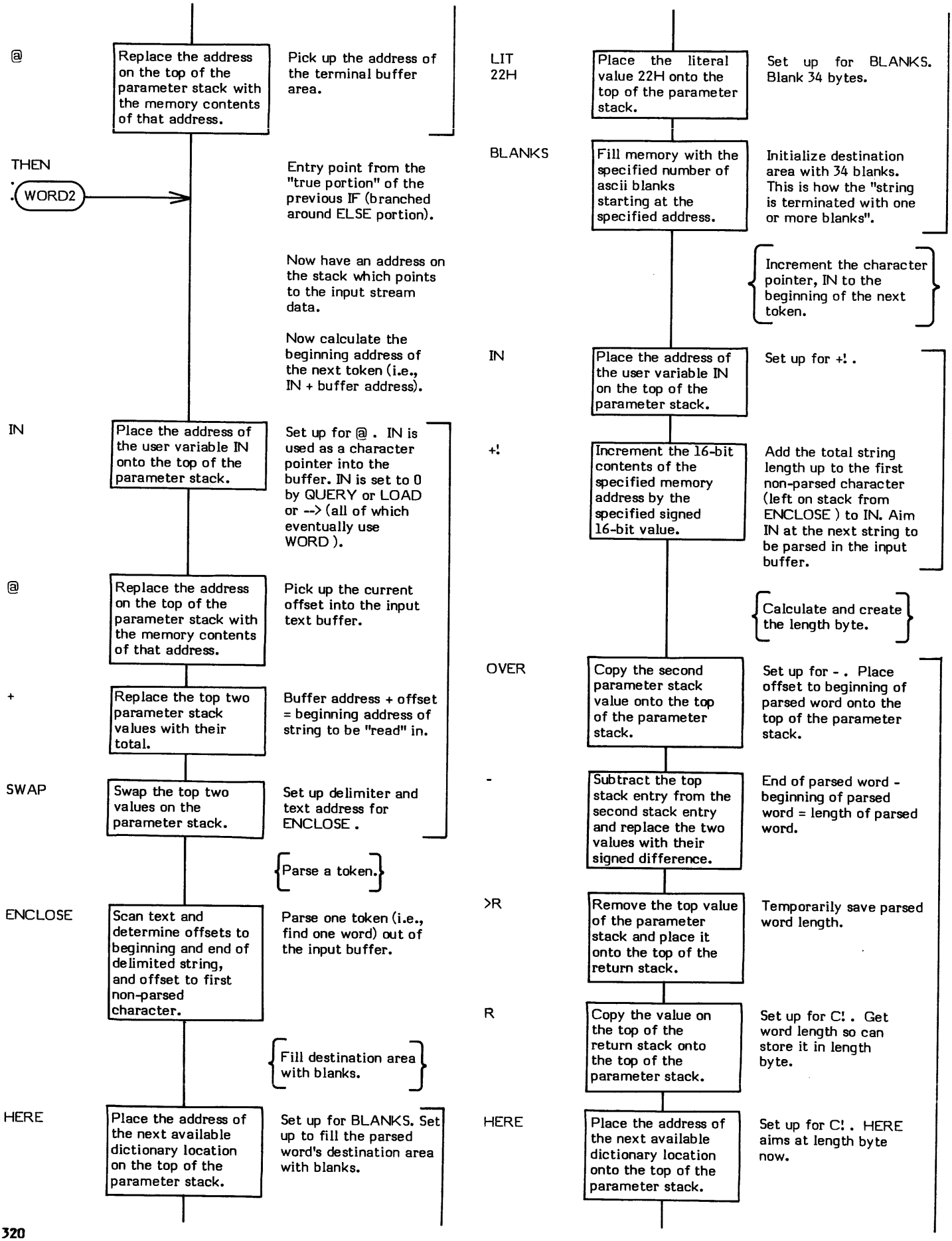
Refer to ENCLOSE , BLK , and IN .

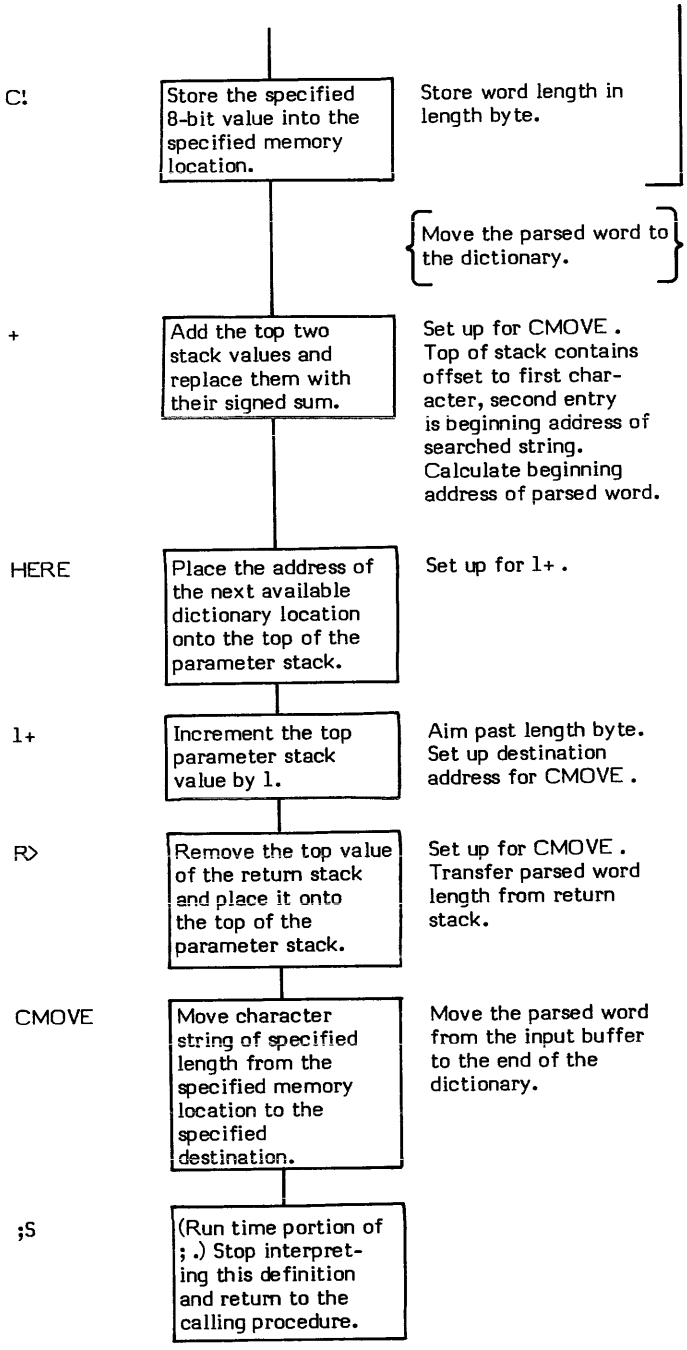
FORTH-79: The FORTH-79 equivalent for WORD is WORD . Note that FORTH-79 requires that WORD return the beginning address of the parsed string.

Definition:

```
:  WORD  ( delimiter -- )
     BLK @        IF
                     BLK @ BLOCK
                  ELSE
                     TIB @
                  THEN
   IN @ + SWAP   ENCLOSE   HERE 22 BLANKS
   IN +!   OVER - >R R HERE C!
   + HERE 1+ R> CMOVE    ;
```




@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the address of the terminal buffer area.
THEN
WORD2
Entry point from the "true portion" of the previous IF (branched around ELSE portion).
Now have an address on the stack which points to the input stream data.
Now calculate the beginning address of the next token (i.e., IN + buffer address).
IN
Place the address of the user variable IN onto the top of the parameter stack.
Set up for @ . IN is used as a character pointer into the buffer. IN is set to 0 by QUERY or LOAD or --> (all of which eventually use WORD).
@
Replace the address on the top of the parameter stack with the memory contents of that address.
Pick up the current offset into the input text buffer.
+
Replace the top two parameter stack values with their total.
Buffer address + offset = beginning address of string to be "read" in.
SWAP
Swap the top two values on the parameter stack.
Set up delimiter and text address for ENCLOSE .
Parse a token.
ENCLOSE
Scan text and determine offsets to beginning and end of delimited string, and offset to first non-parsed character.
Parse one token (i.e., find one word) out of the input buffer.
Fill destination area with blanks.
HERE
Place the address of the next available dictionary location on the top of the parameter stack.
Set up for BLANKS. Set up to fill the parsed word's destination area with blanks.
LIT 22H
Place the literal value 22H onto the top of the parameter stack.
Set up for BLANKS. Blank 34 bytes.
BLANKS
Fill memory with the specified number of ascii blanks starting at the specified address.
Initialize destination area with 34 blanks. This is how the "string is terminated with one or more blanks".
Increment the character pointer, IN to the beginning of the next token.
IN
Place the address of the user variable IN on the top of the parameter stack.
Set up for +! .
+!
Increment the 16-bit contents of the specified memory address by the specified signed 16-bit value.
Add the total string length up to the first non-parsed character (left on stack from ENCLOSE) to IN. Aim IN at the next string to be parsed in the input buffer.
Calculate and create the length byte.
OVER
Copy the second parameter stack value onto the top of the parameter stack.
Set up for - . Place offset to beginning of parsed word onto the top of the parameter stack.
-
Subtract the top stack entry from the second stack entry and replace the two values with their signed difference.
End of parsed word - beginning of parsed word = length of parsed word.
>R
Remove the top value of the parameter stack and place it onto the top of the return stack.
Temporarily save parsed word length.
R
Copy the value on the top of the return stack onto the top of the parameter stack.
Set up for C! . Get word length so can store it in length byte.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
Set up for C! . HERE aims at length byte now.


C!
Store the specified 8-bit value into the specified memory location.
Store word length in length byte.
Move the parsed word to the dictionary.
+
Add the top two stack values and replace them with their signed sum.
Set up for CMOVE . Top of stack contains offset to first character, second entry is beginning address of searched string. Calculate beginning address of parsed word.
HERE
Place the address of the next available dictionary location onto the top of the parameter stack.
Set up for 1+ .
1+
Increment the top parameter stack value by 1.
Aim past length byte. Set up destination address for CMOVE .
R>
Remove the top value of the return stack and place it onto the top of the parameter stack.
Set up for CMOVE . Transfer parsed word length from return stack.
CMOVE
Move character string of specified length from the specified memory location to the specified destination.
Move the parsed word from the input buffer to the end of the dictionary.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.

X

X (— **data address**)

X is a pseudonym for NULL .

Refer to NULL .

FORTH-79: There is no FORTH-79 equivalent for X .

XOR (value1 \ value2 — logical result)

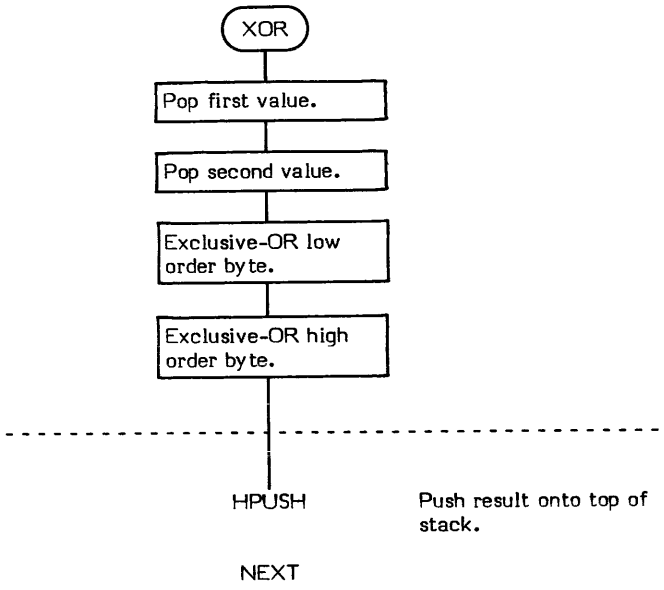
XOR (pronounced "Exclusive-OR" or "X-OR" or "X-O-R") performs a bit-wise logical Exclusive-OR function on the two values on the top of the stack; and replaces them with their logical result.

TOGGLE is an example of a word which uses XOR .

* **At entry** - The top two stack values contain 16-bit logical values to be Exclusive-ORed.
* **At exit** - The top of the stack contains the 16-bit logical result.

XOR is a low level code primitive.

FORTH-79: The FORTH-79 equivalent for XOR is XOR .



[

[(–)

[(pronounced "left-bracket") is used to suspend compilation within a colon definition. The words that follow [are executed, not compiled. Compilation is resumed via the] (resume compilation) command.

An example of the use of [can be found in ;CODE where the assembly language code following ;CODE must be executed and not compiled. [is used within ;CODE to set STATE to 0 (i.e., stop compiling and start interpreting). The assembly language mnemonics following ;CODE are then interpreted which then causes machine code to be compiled into the definition being created.

QUIT uses [to set the system back into interpretation state.

The effect of [is to put a 0 into the user variable STATE .

Note that [is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

* **At entry** - No parameters.
* **At exit** - No parameters.

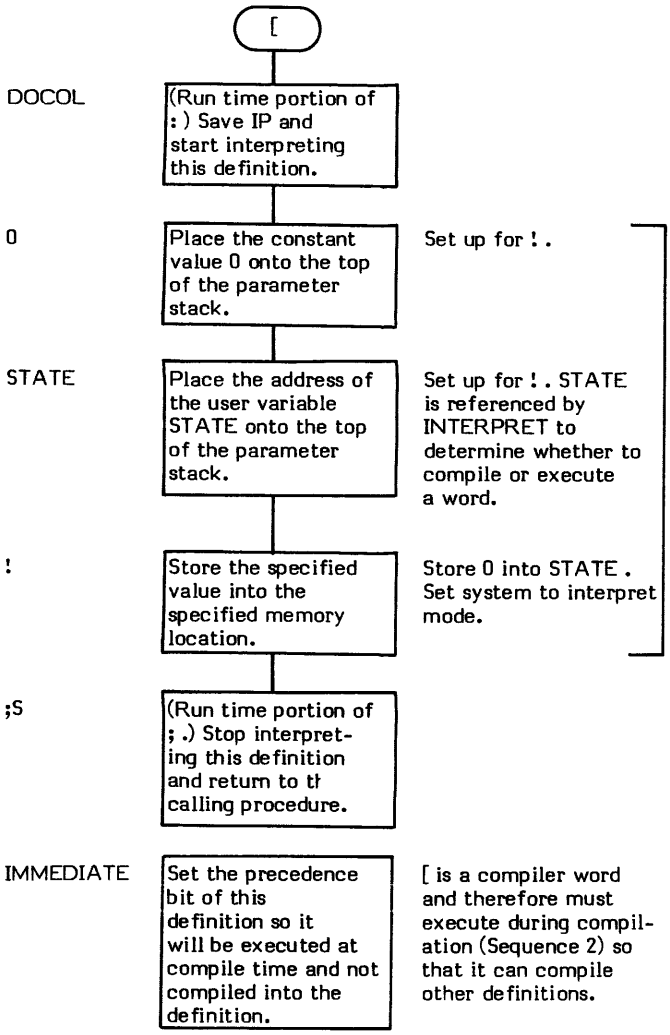
[is a high level colon definition.

Refer to] , INTERPRET , ;CODE , and LITERAL .

FORTH-79: The FORTH-79 equivalent for [is [.

Definition:

```
: [ ( – )
    0 STATE ! ; IMMEDIATE
```



[COMPILE] (—) (Word to compile must be the next word in the input stream)

[COMPILE] (pronounced "bracket-compile") forces the compilation of the immediate word following [COMPILE] in the input stream.

[COMPILE] forces the next input stream word to always be compiled. (i.e., The input stream word immediately following [COMPILE] is compiled into the dictionary and is not executed.) This is the only way immediate words can be compiled (during Sequence 2) so that they can execute at a later time (their Sequence 3).

QUIT is an example of a word which uses [COMPILE] .

[COMPILE] is the functional equivalent of:

[	Stop compiling.
' name	Locate the specified definition in the dictionary.
CFA	Get the definition's CFA .
,	Store the CFA into the dictionary.
]	Start compiling.

* **At entry** - No parameters. The name of the definition to be compiled must immediately follow [COMPILE] in the input stream.
* **At exit** - No parameters.

LIKELY ERROR MESSAGES:

? pronounced "HUH?" (0) -- The word in question cannot be found in the dictionary.

COMPILATION ONLY (11H) -- This word may only be used within a colon definition.

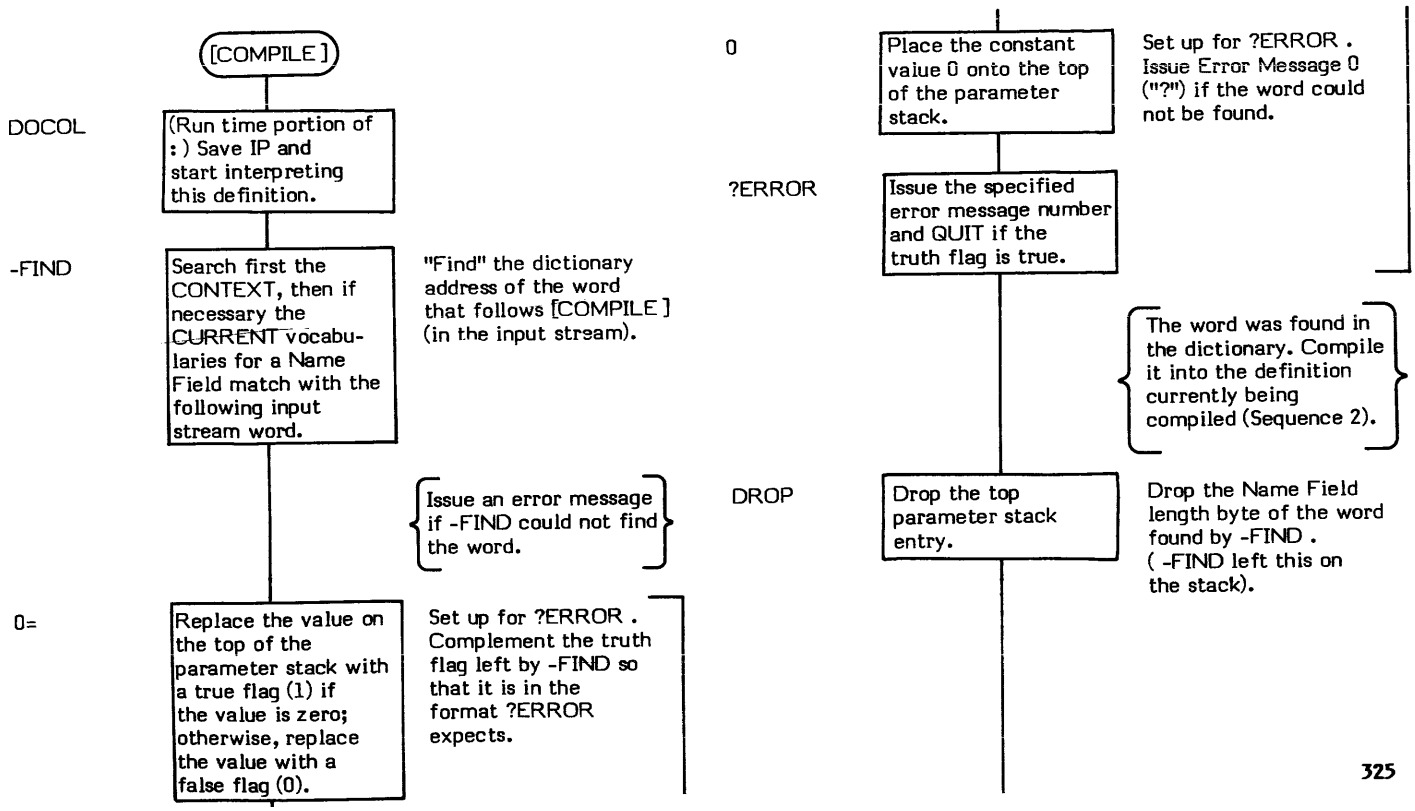
Note that [COMPILE] is an IMMEDIATE word. This means that its precedence bit is set and it will therefore execute at compile time.

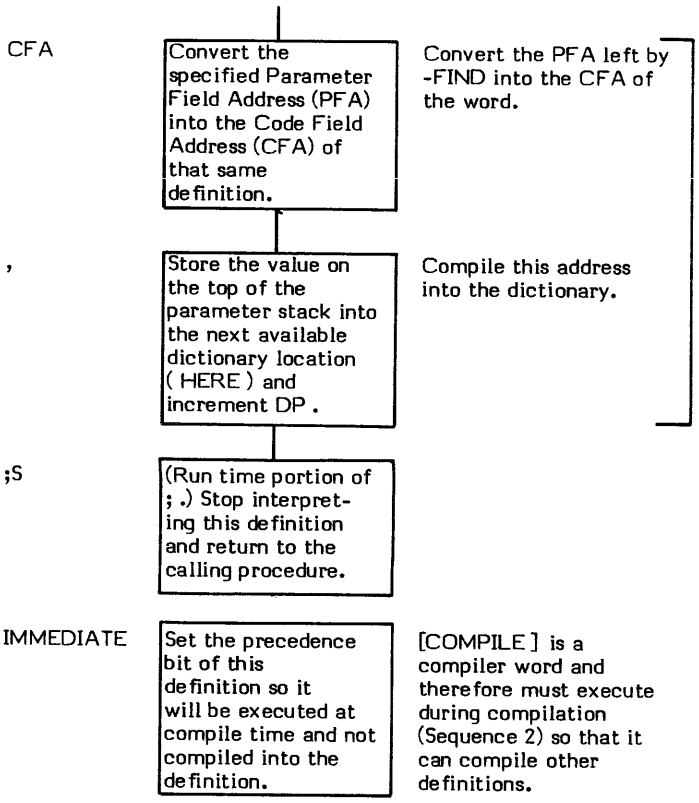
[COMPILE] is a high level colon definition.

Refer to [, COMPILE ,] , and INTERPRET .

FORTH-79: The FORTH-79 equivalent for [COMPILE] is [COMPILE] .

Definition: : [COMPILE] -FIND 0= 0 ?ERROR DROP CFA , ; IMMEDIATE




CFA
Convert the specified Parameter Field Address (PFA) into the Code Field Address (CFA) of that same definition.
Convert the PFA left by -FIND into the CFA of the word.
,
Store the value on the top of the parameter stack into the next available dictionary location (HERE) and increment DP .
Compile this address into the dictionary.
;S
(Run time portion of ; .) Stop interpreting this definition and return to the calling procedure.
IMMEDIATE
Set the precedence bit of this definition so it will be executed at compile time and not compiled into the definition.
[COMPILE] is a compiler word and therefore must execute during compilation (Sequence 2) so that it can compile other definitions.

] (–)

] (pronounced "right-bracket") places the system into the compilation state. This word is normally used to resume compilation after [was used to halt compilation. The effect of] is to place a non-zero value into the user variable STATE .

Note that STATE is set to C0H and not just 01. This is done to save processing time in INTERPRET . Setting STATE to a non-zero value puts the system into compilation mode but, even in compilation mode, IMMEDIATE words must be executed and not compiled (e.g., compiler words). The purpose of using C0H, then, is to cause compilation of all but IMMEDIATE words.

In fig-FORTH this works very simply. A definition is flagged as IMMEDIATE when the 40H bit (the "precedence" bit) in the length byte is set to 1. The most significant bit in the length byte is always set to 1 (the 80H bit). Additionally, the actual length of a name is always greater than 0. Just the logical combination of the precedence bit and the "beginning of Name Field bit" equals C0H. Adding the name length always makes this value greater than C0H because names cannot be 0 characters long.

What INTERPRET does then, is compare the value of the length byte with the contents of STATE . Whether compilation or interpretation takes place is based on the following:

MODE	STATE CONTAINS	ACTION
Interpret	0	All words will have length bytes greater than 0 and INTERPRET will execute the word.
Compile	C0H	IMMEDIATE words will have length bytes greater than C0H and will be executed.
Compile	C0H	Words whose precedence bits are not set will be less than C0H and will be compiled.

In this way, IMMEDIATE words will be executed and non-IMMEDIATE words will be compiled.

An example of a use of] would be to resume compilation after stopping compilation with [, then calculating some value and leaving it on the stack for LITERAL :

: AWORD ----- [calculate value] LITERAL ------ ,

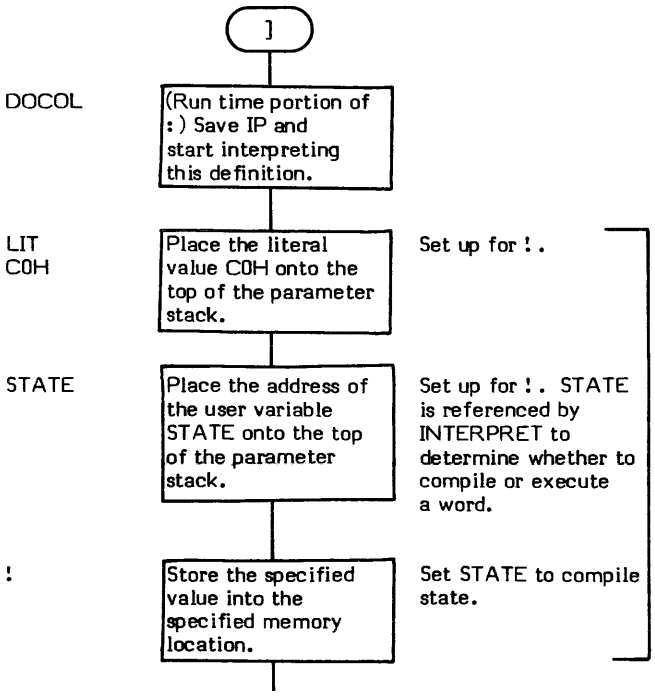
: uses] to set the system to compilation mode so the defined word can be compiled.

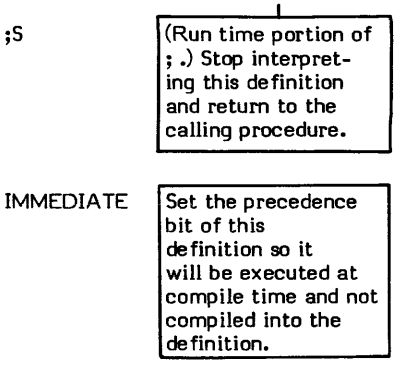
* **At entry** - No parameters.
* **At exit** -No parameters.

] is a high level colon definition.

FORTH-79: The FORTH-79 equivalent for] is] .

Definition: :] (--)
C0 STATE ! ; IMMEDIATE





] is a compiler word and therefore must execute during compilation (Sequence 2) so that it can compile other definitions.

FORTH SYSTEM MESSAGES

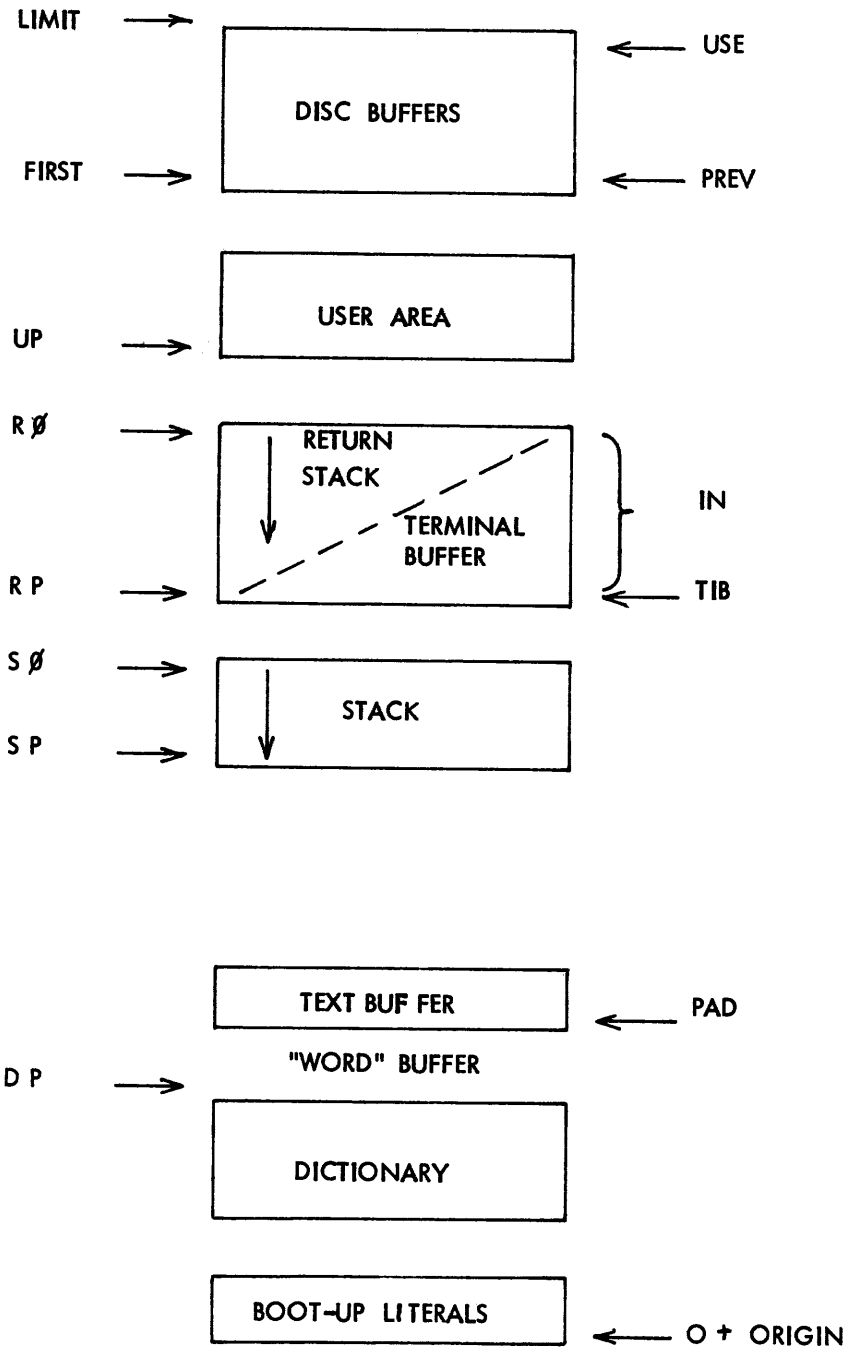
(Relative to Drive 0 Screen 4 Line 0)

HEX	DECIMAL	MESSAGE	DESCRIPTION
0	0	? (pronounced "HUH?")	The word in question cannot be found in the dictionary.
1	1	EMPTY STACK	More values have been removed from the parameter stack than were added. The stack pointer has backed up beyond the initial stack pointer location.
2	2	DICTIONARY FULL	The dictionary has grown into the Terminal Input Buffer.
4	4	ISN'T UNIQUE	The name of this definition already exists elsewhere in the dictionary.
6	6	DISK RANGE?	A disk access to a physically non-existent block number was requested. (Make sure you are not in base hex when you think you are in base decimal.)
7	7	FULL STACK	Too many values have been added to the parameter stack. The stack pointer has gone beyond its upper limit.
8	8	DISK ERROR!	An I/O error occurred while attempting to read or write to virtual I/O. (Make sure diskette is in the drive and the door is shut.)
11	17	COMPILATION ONLY	This word must only be used within a colon definition.
12	18	EXECUTION ONLY	The word must not be used while the system is in compile mode.
13	19	CONDITIONALS NOT PAIRED	There is some sort of problem with the pairing of conditionals within the definition being compiled.
14	20	DEFINITION NOT FINISHED	The position of the parameter stack pointer differs from what it was when this definition began compiling. Something is wrong wtih the definition.
15	21	PROTECTED DICTIONARY	The address of the definition being "forgotten" is less than the value stored in FENCE . Change the value in FENCE .
16	22	USE ONLY WHEN LOADING	This definition should only be used when loading.
17	23	OFF CURRENT EDITING SCREEN	Occurs when using fig-FORTH editor.
18	24	DECLARE VOCABULARY	CONTEXT and CURRENT are not aiming at the same vocabulary when attempting to FORGET . (The purpose of vocabularies is to limit the scope of a definition, so forcing both CONTEXT and CURRENT to be equal prevents inadvertently forgetting a definition.)

ASCII CHARACTER SET (7-BIT CODE)

CODE	CHAR	CODE	CHAR	CODE	CHAR	CODE	CHAR
00	NUL	20	b	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	TAB	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

STANDARD fig-FORTH MEMORY MAP



ALPHABETICAL INDEX

!	1	BL	102	LIT	216
!CSP	2	BLANKS	103	LITERAL	217
#	3	BLK	104	LOAD	219
#>	5	BLOCK	105	LOOP	221
#S	6	BRANCH	109	M*	223
'	7	BUFFER	110	M/	224
(	9	C!	114	M/MOD	226
(.")	10	C,	115	MAX	228
(+LOOP)	12	C/L	116	MESSAGE	229
(;CODE)	13	C@	117	MIN	231
(ABORT)	14	CFA	118	MINUS	232
(DO)	15	CMOVE	119	MOD	233
(FIND)	16	COLD	120	MON	234
(LINE)	19	COMPILE	122	NEXT	235
(LOOP)	21	CONSTANT	124	NFA	236
(NUMBER)	22	CONTEXT	126	NULL	237
*	25	COUNT	127	NUMBER	240
*/	26	CR	128	OFFSET	244
*/MOD	27	CREATE	129	OR	245
+	28	CSP	132	OUT	246
+!	29	CURRENT	133	OVER	247
+-	30	D+	134	PAD	248
+BUF	31	D+-	135	PFA	249
+LOOP	33	D.	136	PREV	250
+ORIGIN	35	D.R	137	QUERY	251
,	37	DABS	139	QUIT	252
-	38	DECIMAL	140	R	254
-->	39	DEFINITIONS	141	R#	255
-DUP	41	DIGIT	142	R/W	256
-FIND	42	DLITERAL	144	R0	259
-TRAILING	44	DMINUS	146	R>	260
.	46	DO	147	REPEAT	261
."	47	DOES>	149	ROT	264
.LINE	49	DP	153	RP!	265
.R	50	DPL	154	RP@	266
/	51	DPUSH	155	S->D	267
/MOD	52	DR0	156	S0	268
0	53	DR1	156	SCR	269
0K	54	DROP	157	SIGN	270
0=	55	DUP	158	SMUDGE	271
0BRANCH	56	ELSE	159	SP!	272
1	57	EMIT	162	SP@	273
1+	58	EMPTY-BUFFERS	163	SPACE	274
2	59	ENCLOSE	164	SPACES	275
2+	60	END	168	STATE	276
3	61	ENDIF	170	SWAP	277
:	62	ERASE	172	TASK	278
;	65	ERROR	173	THEN	279
;CODE	67	EXECUTE	175	TIB	281
;S	70	EXPECT	176	TOGGLE	282
<	71	FENCE	180	TRAVERSE	283
<#	72	FILL	181	TRIAD	285
<BUILDS	73	FIRST	182	TYPE	287
=	76	FLD	183	U*	289
>	77	FLUSH	184	U.	291
>R	78	FORGET	186	U/	292
?	79	FORTH	188	UNTIL	295
?COMP	80	HERE	189	UPDATE	297
?CSP	81	HEX	190	USE	298
?ERROR	82	HLD	191	USER	299
?EXEC	83	HOLD	192	VARIABLE	301
?LOADING	84	HPUSH	193	VLIST	303
?PAIRS	85	I	194	VOC-LINK	305
?STACK	86	ID.	195	VOCABULARY	306
?TERMINAL	88	IF	197	W	314
@	89	IMMEDIATE	199	WARNING	315
ABORT	90	IN	200	WHILE	316
ABS	91	INDEX	201	WIDTH	318
AGAIN	92	INTERPRET	203	WORD	319
ALLOT	94	IP	208	X	322
AND	95	KEY	209	XOR	323
B/BUF	96	LATEST	210	[	324
B/SCR	97	LEAVE	211	[COMPILE]	325
BACK	98	LFA	212	]	327
BASE	99	LIMIT	213		
BEGIN	100	LIST	214		

FUNCTIONAL INDEX

STACK MANIPULATION

-DUP 41
0 53
1 57
2 59
3 61
>R 78
?STACK 86
BL 102
DROP 157
DUP 158
OVER 247
R 254
R0 259
R> 260
ROT 264
RP! 265
RP@ 266
S0 268
SP! 272
SP@ 273
SWAP 277

NUMBER BASES

BASE 99
DECIMAL 140
HEX 190

COMPARISON

0< 54
0= 55
< 71
= 76
> 77

MEMORY

! 1
+! 29
? 79
@ 89
BLANKS 103
C! 114
C@ 117
CMOVE 119
ERASE 172
FILL 181

CONTROL STRUCTURE

(+LOOP) 12
(DO) 15
(LOOP) 21
+LOOP 33
0BRANCH 56
AGAIN 92
BEGIN 100
BRANCH 109
DO 147
ELSE 159
END 168
ENDIF 170
I 194
IF 197
LEAVE 211
LOOP 221
REPEAT 261
THEN 279
UNTIL 295
WHILE 316

TERMINAL INPUT/OUTPUT

(.") 10
(LINE) 19
. 46
." 47
.LINE 49
.R 50
?TERMINAL 88
C/L 116
COUNT 127
CR 128
D. 136
D.R 137
EMIT 162
EXPECT 176
IN 200
KEY 209
MESSAGE 229
QUERY 251
SPACE 274
SPACES 275
TIB 281
TRIAD 285
TYPE 287
U. 291
VLIST 303
WORD 319

INPUT-OUTPUT FORMATTING

3
#> 5
#S 6
(NUMBER) 22
-TRAILING 44
<# 72
COUNT 127
CR 128
DIGIT 142
DPL 154
ENCLOSE 164
FLD 183
HLD 191
HOLD 192
NUMBER 240
OUT 246
PAD 248
R# 255
SIGN 270

VOCABULARIES

CONTEXT 126
CURRENT 133
DEFINITIONS 141
FORTH 188
HERE 189
LATEST 210
VOC-LINK 305
VOCABULARY 306

COMPILER SECURITY

!CSP 2
?COMP 80
?CSP 81
?ERROR 82
?EXEC 83
?LOADING 84
?PAIRS 85
?STACK 86
CSP 132
ERROR 173
SMUDGE 271
WARNING 315

DICTIONARY

' 7
(FIND) 16
+ORIGIN 35
-FIND 42
ALLOT 94
CFA 118
CONTEXT 126
CURRENT 133
DEFINITIONS 141
DP 153
FENCE 180
FORGET 186
FORTH 188
ID. 195
LATEST 210
LFA 212
NFA 236
PFA 249
SMUDGE 271
TRAVERSE 283
VLIST 303
VOC-LINK 305
WIDTH 318

ARITHMETIC

* 25
*/ 26
*/MOD 27
+ 28
+! 29
+- 30
- 38
/ 51
/MOD 52
1+ 58
2+ 60
ABS 91
D+ 134
D+- 135
DABS 139
M* 223
M/ 224
M/MOD 226
MAX 228
MIN 231
MOD 233
S->D 267
U* 289
U/ 292

LOGICAL

AND 95
DMINUS 146
MAX 228
MIN 231
MINUS 232
OR 245
TOGGLE 282
XOR 323

VIRTUAL INPUT-OUTPUT

+BUF 31
--> 39
B/BUF 96
B/SCR 97
BLK 104
BLOCK 105

BUFFER 110
DR0 156
DR1 156
EMPTY-BUFFERS 163
FIRST 182
FLUSH 184
INDEX 201
LIMIT 213
LIST 214
LOAD 219
OFFSET 244
PREV 250
R/W 256
SCR 269
UPDATE 297
USE 298

COMPILER DEFINING WORDS

(9
(;CODE) 13
, 37
0BRANCH 56
: 62
; 65
;CODE 67
;S 70
<BUILDS 73
BACK 98
BRANCH 109
C, 115
COMPILE 122
CONSTANT 124
CREATE 129
DLITERAL 144
DOES> 149
IMMEDIATE 199
LIT 216
LITERAL 217
SMUDGE 271
STATE 276
VARIABLE 301
VOCABULARY 306
[324
[COMPILE] 325
] 327

SYSTEM WORDS

(ABORT) 14
ABORT 90
COLD 120
EXECUTE 175
IP 208
MON 234
PAD 248
QUIT 252
TASK 278
TIB 281
USER 299

INNER INTERPRETER

DPUSH 155
HPUSH 193
IP 208
NEXT 235
W 314

OUTER INTERPRETER

INTERPRET 203
NULL 237
QUERY 251
QUIT 252
STATE 276
X 322
[324

] 327