

KITT PEAK NATIONAL OBSERVATORY

MEMORANDUM

TO Distribution Date October 1979

FROM R. Stevens SUBJECT Forth Primer - Update 3

✓ Attached is Update 3 of the Forth Primer. This update is a complete reprinting of this manual. Please discard any old copies of the Primer that you may have.

The major changes to the Primer are:

- Correspondence with KPNO Forth, Versions 3.0 and later. This level of KPNO Forth conforms (in general) to the 1978 Forth International Standard;
- Description of the file system (Section 13.2). This section corresponds to KPNO Forth, Versions 3.3 and later;
- Description of overlays (Section 13.3);
- Description of vocabularies (Section 13.4).

As usual, any comments or suggestions concerning any aspect of the primer are welcome.

PROPERTY OF THE U. S. ARMY
REDSTONE SCIENTIFIC INFORMATION CENTER
REDSTONE ARSENAL, ALABAMA

KITT PEAK NATIONAL OBSERVATORY *

Tucson, Arizona 85726

A F O R T H P R I M E R

PROPERTY OF THE U. S. ARMY
REDSTONE SCIENTIFIC INFORMATION CENTER
REDSTONE ARSENAL, ALABAMA

W. Richard Stevens
October 1979
(Update 3)

*Kitt Peak National Observatory is operated by the Association of Universities for Research in Astronomy, Inc., under contract with the National Science Foundation.

2A72.6
K62
1979
C1
X

TABLE OF CONTENTS

1.	Introduction1-1
2.	Obtaining a copy of the current FORTH system2-1
3.	Loading FORTH into the computer.3-1
3.1	Running from disc.3-1
3.2	Running from tape.3-2
3.3	Restarting from disc3-3
3.4	Saving your program modifications.3-6
4.	Executing FORTH utility words.4-1
4.1	Terminal Interaction4-1
4.2	Re-formatting the disc4-3
4.3	Listing FORTH blocks4-4
5.	Arithmetic Expressions5-1
5.1	The FORTH Stack.5-1
5.2	Infix/Polish Notation.5-4
6.	The FORTH Dictionary6-1
7.	Data Structures.7-1
7.1	Integers7-1
7.2	Double-word Integers7-6
7.3	Floating-point Numbers7-11
7.4	Conversions between Data Structures.7-18
7.5	Logical Values and Logical Expressions7-20
7.6	Additional Numeric Conversions7-22
7.7	Vectors.7-25
8.	Stack Operations8-1
8.1	Manipulation Words8-1
8.2	Comparison Words8-9
9.	The Colon Definition9-1

10.	Program Control	10-1
10.1	DO LOOPS	10-1
10-2	BEGIN-END Loops.	10-7
10-3	BEGIN-WHILE-REPEAT Loops	10-9
10-4	IF-THEN-ELSE Statement Selection	10-10
11.	Block I/O	11-1
12.	Text Editor	12-1
12.1	Special Characters and Terminology	12-1
12.2	Command Descriptions	12-2
12.3	Block Editor	12-13
13.	Program Structure	13-1
13.1	Block Oriented Programs.	13-1
13.2	File System.	13-11
13.3	Overlays	13-18
13.4	Vocabularies	13-22
14.	Terminal I/O.	14-1
14.1	Character Output	14-1
14.2	Numeric Input.	14-2
14.3	Numeric Output	14-3
15.	Advanced Arithmetic	15-1
15.1	Numerical Functions.	15-1
15.2	Mixed Precision Operators.	15-7
15.3	Arithmetic Range Errors.	15-9
15.4	Combined Words	15-10
16.	Real-Time I/O	16-1
16.1	Interrupts	16-1
16.2	CAMAC I/O.	16-2
16.3	FORTH CAMAC Words.	16-5
16.4	FORTH Interrupt Words.	16-10

Appendices

A.	ASCII Character Set	A-1
B.	FORTH Error Codes	B-1
C.	Answers to Exercises.	C-1
D.	FORTH Glossary.	D-1

1. INTRODUCTION

FORTH is a programming system whose main function is to simplify the programming of minicomputers that are used for on-line data acquisition. This primer is intended as an introduction to FORTH. The only assumption made is that the reader has some experience and acquaintance with computers in general (probably through the FORTRAN language) and is capable of pursuing a self study course.

The organization of this primer is to present the features of FORTH in an orderly and stepwise fashion. This primer should be read in the order presented as each chapter builds on the points covered in the previous chapter.

Computer Science is a field in which "hands on" experience is a requisite; that is, one can read a plethora of programming manuals and obtain some information, however one *must* write and debug some programs using a given programming tool in order to really understand and appreciate the tool being used. FORTH is a classical example of this principle and a perusal of this primer without trying to work or understand the examples and exercises will yield very little knowledge of FORTH. This requisite for "hands on" experience is especially true with FORTH since it is an interactive, terminal-oriented, minicomputer system quite different from the batch-oriented FORTRAN systems you are probably familiar with.

This manual is intended to be used as a self study tool and therefore exercises to be worked, along with *complete* solutions are provided. It must be clearly stated that the provided solutions are not to be considered the *only* solution to an exercise. Programming is an art and therefore there will usually exist more than one solution to a given problem. The analysis of all possible solutions, in order to determine the "best" solution, is not a clearly defined task, mainly due to the variable criteria available to specify which solution is "best". The reader should not be disturbed if he arrives at a solution that is not identical with the provided solution but should try to understand the provided solution (and possibly compare the two solutions to determine which, if either, is "better").

As with any self study course, referral to the solution for an exercise, before making an honest attempt to solve the exercise on your own, defeats the purpose of this primer.

Although an occasional reference is made to the version of FORTH used at Kitt Peak (for its Varian 620 minicomputers) this primer is largely independent of any specific FORTH implementation. In fact, until the final chapter no mention is made of the computer being binary or decimal and no specification of the number of bits in a computer word is needed.

This manual is a primer and as such does *not* describe FORTH in its entirety. All of the nitty gritty details of the implementation of FORTH are ignored, instead this primer tries to give the reader an appreciation of what FORTH is and how to use it to solve a large class of problems on a minicomputer. The most notable exclusion from this primer is a description of machine language programming in FORTH, this topic being so dependent on the specific computer being used. This topic along with an advanced description of the implementation of FORTH is covered in "FORTH - Systems Reference Manual" which is available from the author at Kitt Peak National Observatory, Tucson, Arizona 85726.

Comments and suggestions concerning any portion of this manual are solicited. Please try to be as specific as possible (reference the page number and revision level). Direct all comments to the author.

"Explain all that," said the Mock Turtle.

"No, no, the adventures first," said the gryphon in an impatient tone: "Explanations take such a dreadful time."

LEWIS CARROLL

Alice's Adventures in Wonderland

NOTE: The version of FORTH used throughout this manual is Kitt Peak FORTH, Versions 3.0 and later, which runs on a Varian 620 minicomputer. This version of FORTH corresponds to the basic system defined by the FORTH International Standards Team along with many Kitt Peak extensions.

2. OBTAINING A COPY OF THE CURRENT FORTH SYSTEM

The first thing one must do is obtain a copy of the current version of the KPNO Varian FORTH system on a magnetic tape. This tape may then be taken to any of the KPNO Varian systems, loaded into the system and executed.

The current version of FORTH will always reside on the Kitt Peak CDC 6400 and the following job will copy the system onto your magnetic tape (a 600 foot tape is sufficient).

jobname,account#,MT1.	jobname and account# are parameters required by the 6400 SCOPE operating system.
VSN(TAPE9= tape#)	tape# is the volume serial number or volume serial name on the magnetic tape.
REQUEST(TAPE9,HI,S,RING)	causes the mag tape to be mounted on a tape drive with a write ring and directs the system to write the tape at 556 bpi.
ATTACH(TAPE5,DOFORTH,CY=5)	attaches the current version of FORTH to tape5.
ATTACH(DOFORTH,DOFORTH)	attaches the public file DOFORTH which is the FORTH utility program used to manipulate FORTH tapes.
DOFORTH.	execute DOFORTH.
end-of-record card	terminates the SCOPE commands.
end-of-record card	terminates the \$INTYPE cards.
\$OUTTYPE TAPE=.TRUE., IBLK1=1, IBLK2=199 \$ ↑ col. 2 of card	
end-of-file card	terminates the \$OUTTYPE cards and terminates the program.

Upon successful execution of this job the user should remove the write ring from the magnetic tape. The contents of the tape (which will automatically be printed by the above job) are blocks 1-199 of the current FORTH system (blocks 1-7 are not printed).

If the user has a FORTH tape that he wants listed on the CDC 6400 (Section 3.4 describes the procedure for creating a FORTH tape on the minicomputer) the following job may be used:

jobname,account#,MT1.	jobname and account# are parameters required by the 6400 SCOPE operating system.
VSN(TAPE8= tape#)	tape# is the volume serial number or volume serial name on the magnetic tape to be listed.
REQUEST(TAPE8,HI,S)	causes the mag tape to be mounted on a tape drive without a write ring and directs the system to read the tape at 556 bpi.
ATTACH(DOFORTH,DOFORTH)	attaches the public file DOFORTH which is the FORTH utility program used to manipulate FORTH tapes.
DOFORTH.	execute DOFORTH.

end-of-record card

terminates the SCOPE commands.

\$INTYPE TAPE=.TRUE. \$

↑
└ column 2 of card

end-of-record card

terminates the \$INTYPE commands.

\$OUTTYPE TAPE=.FALSE. \$

↑
└ column 2 of card

end-of-file card

terminates the \$OUTTYPE commands
and terminates the program.

This job will read in blocks 1-511 from the user's FORTH tape and then produce a line printer listing of these blocks.

3. LOADING FORTH INTO THE COMPUTER

Now that you have a copy of FORTH on tape the steps required to load the tape into a computer and then execute the FORTH system must be described. FORTH is somewhat unique in that the system will run from either a disc or a tape and each process is described separately below.

3.1 RUNNING FROM DISC

- 1) Power on the computer and all associated peripherals.
- 2) Mount your FORTH tape on the magnetic tape drive and check that the switches on the magnetic tape controller (located immediately above the tape drive) are set as follows:

Density	HI/LOW	->	LOW
Mode	REMOTE/MANUAL	->	MANUAL
Parity	EVEN/ODD	->	ODD

Position the tape to the load point (push the load button twice) then place the drive on-line. At this point the LOAD and ONLINE buttons should be lighted.

- 3) Place the disc STOP/READY switch to the READY position and wait approximately 40 seconds for the READY light to come on.
- 4) Set the Sense Switches on the CPU as follows:

Sense Switch 1	->	DOWN	(load from tape)
Sense Switch 2	->	UP	(see note below)
Sense Switch 3	->	UP	(run from disc)

- 5) Press the following switches on the CPU:

STEP/RUN	->	STEP
RESET	->	Press down
STEP/RUN	->	RUN
BOOTSTRAP	->	Press down

Note: Sense Switch 2 is looked at only if you are loading from tape onto disc. In this case, if Sense Switch 2 is DOWN, then the region of disc that will be copied from the tape is zeroed before the tape is copied onto disc. If Sense Switch 2 is up, the disc region is not zeroed. If you plan to make modifications to some program blocks and then save a copy of the new program (Section 3.4), Sense Switch 2 should be DOWN.

At this point the tape will be copied onto the disc and at completion the tape will automatically rewind. The RUN light and the OVFL light on the CPU should be lighted.

In order to load the basic FORTH system into core from the disc press any key on the terminal (such as the RETURN key) and FORTH will respond by ringing the terminal bell. Basic FORTH will automatically be loaded into core and the following message will be output to the terminal:

```
FORTH x.y date      620/F AND DISK
      comment
```

(*x.y* denotes the version of the FORTH system, *date* specifies the creation date of the FORTH system and *comment* is an operator specified comment that describes the contents of the tape (refer to Section 3.4)).

3.2 RUNNING FROM TAPE

- 1) Power on the computer and all associated peripherals.
- 2) Mount your FORTH tape on the magnetic tape drive and check that the switches on the magnetic tape controller (located immediately above the tape drive) are set as follows:

Density	HI/LOW	->	LOW
Mode	REMOTE/MANUAL	->	MANUAL
Parity	EVEN/ODD	->	ODD

Position the tape to the load point (push the load button twice) then place the drive on-line. At this point the LOAD and ONLINE buttons should be lighted.

- 3) Set the sense switches on the CPU as follows:

Sense Switch 1	->	DOWN	(load from tape)
Sense Switch 2	->	UP	
Sense Switch 3	->	DOWN	(run from tape)

4) Press the following switches on the CPU:

```
STEP/RUN  ->  STEP
RESET      ->  press down
STEP/RUN  ->  RUN
BOOTSTRAP ->  press down
```

At this point the tape will briefly (2seconds) move and then the RUN light and the OVFL light on the CPU will be lighted.

In order to load the basic FORTH system into core from the tape press any key on the terminal (such as the RETURN key) and FORTH will respond by ringing the terminal bell. Basic FORTH will automatically be loaded into core and the following message will be output to the terminal.

```
FORTH  x.y date      620/F AND TAPE
        comment
```

(*x.y* denotes the version of the FORTH system, *date* specifies the creation date of the FORTH system and *comment* is an operator specified comment that describes the contents of the tape (refer to Section 3.4)).

3.3 RESTARTING FROM DISC

The two procedures described above are referred to as "cold-start" procedures since they make no assumptions concerning the contents of the disc or core, instead basic FORTH is reloaded from magnetic tape. Obviously this procedure takes some time (depending on how many programs and/or data are contained on the tape) therefore if we are certain that the FORTH system contained on the disc is usable (i.e., the disc has not been erased or overwritten since FORTH was last loaded from tape onto disc) we may save time by loading FORTH from the disc and not reading in the magnetic tape version. This procedure is referred to as a "warm-start" and may be performed whenever the user has clobbered the FORTH system in core (but not the system on disc):

1) Set the Sense Switches on the CPU as follows:

Sense Switch 1 ---> UP (load from disc)

Sense Switch 2 ---> UP

Sense Switch 3 ---> UP (run from disc)

2) Press the following switches on the CPU:

STEP/RUN ---> STEP

RESET ---> press down

STEP/RUN ---> RUN

BOOTSTRAP ---> press down

The RUN light and the OVFL light on the CPU will be lighted. To load the basic FORTH system into core from the disc press any key on the terminal (such as the RETURN key) and FORTH will respond by ringing the terminal bell. Basic FORTH will automatically be loaded into core.

It should be obvious that this "warm-start" procedure is applicable only if FORTH was originally loaded from tape onto the disc. If you are running from tape and wish to reload FORTH you must go through the entire tape load procedure again (Section 3.2).

One additional "warm-start" procedure is available, namely keying in the word ZAP terminated by a carriage-return. FORTH will respond by ringing the terminal bell and Basic FORTH will automatically be loaded into core.

This procedure has the advantage that the entire reloading process is done through the terminal and you do not have to set any switches on the CPU - an obvious benefit if the CPU is separated from the terminal by some distance. The disadvantage of this procedure is that FORTH must be responding to terminal input in order for you to enter and execute the word ZAP. If you have somehow destroyed the FORTH system in core to the point that it is not accepting terminal input then you have to resort to either a disc "warm-start" or a "cold-start".

Load from Tape Run from Disc	Load from Tape Run from Tape	Load from Disc Run from Disc
<p>Power-on all computer equipment.</p> <p>Mount system tape.</p> <p>Sense Switch 1 - DOWN Sense Switch 2 - UP Sense Switch 3 - UP</p> <p>STEP/RUN → STEP RESET → press down STEP/RUN → RUN BOOTSTRAP → press down</p> <p>Entire tape reads in.</p> <p>Press any terminal key to load basic FORTH.</p>	<p>Power-on all computer equipment.</p> <p>Mount system tape.</p> <p>Sense Switch 1 - DOWN Sense Switch 2 - UP Sense Switch 3 - DOWN</p> <p>STEP/RUN → STEP RESET → press down STEP/RUN → RUN BOOTSTRAP → press down</p> <p>First few records on tape read in.</p> <p>Press any terminal key to load basic FORTH.</p>	<p>Power-on all computer equipment.</p> <p>Sense Switch 1 - UP Sense Switch 2 - UP Sense Switch 3 - UP</p> <p>STEP/RUN → STEP RESET → press down STEP/RUN → RUN BOOTSTRAP → press down</p> <p>Press any terminal key to load basic FORTH.</p>

Table 3.1 - Loading FORTH into the Computer

3.4 SAVING YOUR PROGRAM MODIFICATIONS

If, after loading FORTH onto the disc, you make modifications and changes to your program(s) you will want to save a copy of these changes on magnetic tape (since the next person who uses the disc may erase or overwrite your program blocks). The procedure to do this is as follows:

- 1) Mount a scratch tape on the tape drive with a write ring.
- 2) Execute the word `SAVEDISK` and the following will be output to the terminal:

```
ENTER NEW COMMENT OR RETURN, TO SAVE BLOCKS 1-511
OLD COMMENT: ( old comment )
```

Old comment is the comment line that was printed after loading basic FORTH. This comment serves no purpose except to provide the operator with a message identifying the FORTH system and program(s) that were loaded. Each time the disk is saved the operator has the option of changing this comment and if you so desire you may key in a new comment at this point (up to 63 characters, terminated by a carriage-return). If you wish to retain the old comment then simply enter a carriage-return.

- 3) After you have either keyed in a new comment or entered a carriage return one of the following messages will be output to the terminal:

```
TAPE ON LINE.      ** NO MAP READ **      or
TAPE DOES NOT RESPOND.
```

(If the second message is printed then the tape drive is not on line.)

- 4) Blocks 1 through 511 will be copied from disc onto tape and then the tape will be rewound. The following message will be output to the terminal:

*** BLOCKS 1-511 SAVED ***

This tape may now be taken to any of the mini-computer systems and loaded into core using the methods described in Sections 3.1 and 3.2. Additionally this tape may be taken to the CDC 6400 and listed on the line printer using the method described in Chapter 2.

EXERCISES - CHAPTER 3

- 1) Perform a cold-start and run FORTH from disc. How long does the procedure take? Now that FORTH resides on disc perform a warm-start from the disc. Perform a ZAP.
- 2) Perform a cold-start and run FORTH from tape. How long does the procedure take?

4. EXECUTING FORTH UTILITY WORDS

After having learned the procedures required to load FORTH into the computer, the purpose of this chapter is to have you execute and use some system defined routines thereby gaining some feeling for the operator-machine interaction provided by an interactive system such as FORTH.

4.1 TERMINAL INTERACTION

The first concept to understand is the entering and execution of words through the terminal. You are already familiar with this from executing the word ZAP from the previous chapter. The general rules are:

- FORTH does not interpret a line of operator input until the operator terminates the line by entering a *carriage-return*.
- The operator may delete the previous character by entering a *rubout*. FORTH responds by backspacing one character.
- The operator may delete an entire line by entering a Control-U. FORTH responds by printing "\".

After entering a carriage-return to terminate a line of input, FORTH will go through the line and execute every word in the input line.

The definition of a FORTH word is very simple:

A FORTH word is a sequence of up to 64 characters, preceded by a space and terminated by a space. The sequence of characters may contain any character in the ASCII character-set (Appendix A) except carriage-return, rubout, Control-U or space.

For example, entering the line

```
1 HELLO? RESIDENT# ZAP-ZAP
```

(terminated by a carriage-return) will cause FORTH to execute the four words

```
1
HELLO?
RESIDENT#
ZAP-ZAP
```

(The actual execution of each word will be discussed later, presently we are just interested in the entering of words in an input line through the terminal.) The words are executed in the order in which they are entered.

If all goes well and FORTH successfully executes each word in the input string then FORTH responds with a carriage-return, line-feed (i.e. - moves to the beginning of the next line), outputs an asterisk and waits for the operator to enter another line of input. This loop (enter a line of input, execute each word in the input line, ...) is the heart of the FORTH system.

If FORTH detects an error of any sort while executing a word in the input string, FORTH will output the name of the word it was executing when the error was detected, followed by a question-mark and a single character identifying the type of error. (A listing of the single character error codes and a description of each is found in Appendix B). For example, if the operator entered the four words as shown above and for some reason a Q error occurred while executing the word HELLO? then FORTH will output

```
HELLO ?Q
```

Similarly, if a U error occurred while executing the word ZAP-ZAP FORTH will respond with

```
ZAP-ZAP ?U
```

4.2 RE-FORMATting THE DISC

This exercise is a good example of an interactive program. For reasons that are not important here, it occasionally becomes necessary to re-format the disc (this involves the updating of certain timing tracks used by the hardware to access the data on the disc). The disc consists of two platters, a removable platter and a fixed platter, either of which may be re-formatted.

After having loaded basic FORTH into core execute

UTIL FORMATTER

and a list of instructions should be output on the terminal. Execute

R-CHECK

and note any format errors on the removable platters (hopefully there should not be any). Similarly execute

F-CHECK

to check the fixed platter. Then re-format both platters by executing

R-FMT

and subsequently

F-FMT

Both platters may be zero'ed (i.e. - erased) by executing

R-ZERO F-ZERO

Note that this final step (zero'ing the entire disc) erases the FORTH system stored on the disc, necessitating a cold-start from tape the next time you wish to re-load the system.

After completing this little exercise, execute the word

DISCARD

which effectively throws away the last program loaded (the disc re-formatter) so that you may re-use the core that it took up.

4.3 LISTING FORTH BLOCKS

As will be discussed later, FORTH requires the user to break up his programs and data into chunks of storage referred to as "blocks". Each block is identified by a unique number between 0-4895. One may list a FORTH block to see what its contents are: to list block 80 on the terminal, execute

```
80 LIST
```

This may be done for any block.

If the FORTH system that you are using has a Centronix line printer attached to it then execute

```
UTIL PRINTERS CEN
```

```
180 LOAD
```

```
80 86 BLOCKPRINT
```

to list blocks 80 through 86 on the line printer.

5. ARITHMETIC EXPRESSIONS

5.1 THE FORTH STACK

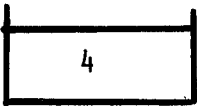
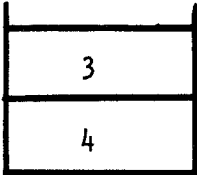
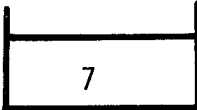
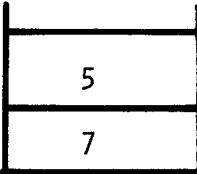
One of the most unique features of FORTH is its use of a pushdown stack (referred to simply as the stack) to hold operands and parameters.

Some examples are the easiest way to describe the use of the stack:

Consider the input line

4 3 + 5 - .

Recall that FORTH will execute each word in the input string, one word at a time, from left to right (refer to Section 4.1). The following actions will take place as FORTH executes each of the six words:

WORD FROM INPUT	ACTION	CONTENTS OF STACK
4	FORTH interprets this word as a number and pushes its value onto the stack ----->	
3	FORTH interprets this word as a number and pushes its value onto the stack ----->	
+	This word is the arithmetic "addition" operator - it expects two numbers to be in the top two positions on the stack and these two numbers are added together. The two operands are then replaced on the stack by the result ----->	
5	FORTH interprets this word as a number and pushes its value onto the stack ----->	

WORD FROM INPUT	ACTION	CONTENTS OF STACK
-	This word is the arithmetic "subtraction" operator - it expects two numbers to be in the top two positions on the stack and the number on top is subtracted from the number below. The two operands are then replaced on the stack by the result ----->	<div style="border: 1px solid black; padding: 5px; display: inline-block;">2</div>
.	This word simply removes the top number from the stack and prints the number on the terminal ----->	stack empty

Two terms were introduced in the above example: *push* and *pop*. To *push* a number onto the stack is to place the number at the top of the stack. To *pop* a number from the stack is to remove the top number from the stack.

The general rules of FORTH's stack manipulation are:

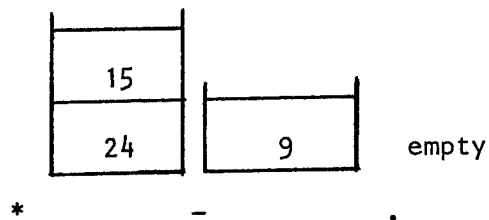
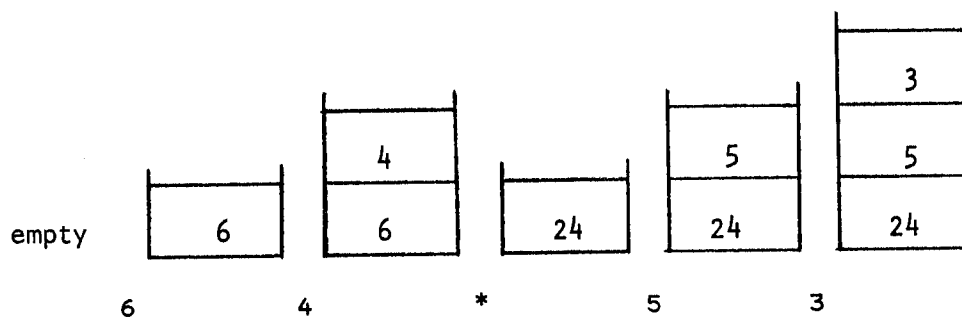
- 1) Any number to be placed on the stack must be pushed onto the top position on the stack. Any number to be removed from the stack must reside at the top of the stack.
- 2) Arithmetic operators expect their operands to be in the top positions of the stack. After completion of the arithmetic operation the operands are popped from the stack and the result is pushed onto the stack.
- 3) General words that operate on numbers residing on the stack (such as the word `.` in the above example to print the top number on the stack) remove from the stack the number operated on. This means,

for example, that the number printed is removed from the stack (as occurred in the above example).

It should be obvious that the size of the stack is dynamic, that is, it changes continually. One of the easiest ways to understand what operation a sequence of numbers and operators performs is to keep track of the contents of the stack. For example, the result of the input line

6 4 * 5 3 * - .

is seen to be 9, from the following stack diagrams:



5.2 INFIX/POLISH NOTATION

The standard representation of a mathematical expression that one is accustomed to (from programming languages such as Fortran, Algol, etc.) is referred to as infix notation. Infix notation requires that an operator be preceded and followed by the two operands that it is to process (assume we are dealing only with binary operators such as +, -, * and /). One limitation of infix notation is that one must specify a hierarchy of precedence among the operators in order to unambiguously handle expressions such as:

$$2 + 3 * 4$$

Does this denote "2 plus the product of 3 and 4" or "4 times the sum of 2 plus 3"? (Fortran would use the first interpretation). One can further complicate the translator of these mathematical expressions by introducing parentheses to explicitly denote the desired ordering of an expression. One could then write the above example as either

$$\begin{array}{lcl} 2 + (3 * 4) & \text{--->} & 14 \\ (2 + 3) * 4 & \text{--->} & 20 \end{array}$$

depending on the desired meaning.

A completely unambiguous representation of the above example may be written in Polish-postfix notation (also referred to as parentheses-free notation):

$$2 \ 3 \ + \ 4 \ *$$

Here the operators follow the operand (hence the adjective "postfix") and the expression may be easily evaluated from left-to-right with the aid of FORTH's stack as follows:

WORD	ACTION REQUIRED	CONTENTS OF STACK
2	number, push it onto the stack.	<div> <div></div> <div>2</div> </div>
3	number, push it onto the stack.	<div> <div>3</div> <div>2</div> </div>
+	operator, take two numbers on top of stack, add them together, delete the top two numbers on stack and replace with the result.	<div> <div></div> <div>5</div> </div>
4	number, push it onto the stack.	<div> <div>4</div> <div>5</div> </div>
*	operator, take two numbers on top of stack, multiply them together, delete the top two numbers on stack and replace with result.	<div> <div></div> <div>20</div> </div>

For completeness it should be noted in passing that one may find references to Polish-prefix notation. In this case the operators precede the operands and the notation is then evaluated from right-to-left. Polish-prefix and Polish-postfix are basically identical and it is usually a matter of preference if the expression is to be evaluated left-to-right or right-to-left, however Polish-postfix is the more prevalent. The above example in Polish-prefix would be

* 5 + 2 3

For the trivia minded it is mentioned that these notations were originally developed by the Polish mathematician Lukasiewicz.

The main advantage of Polish notation over infix notation is the unambiguity in the representation of an expression. The conversion of an expression from infix to Polish (as done, for example, by a Fortran compiler) requires some additional processing. The Polish-postfix notation employed by FORTH transfers this conversion process from the FORTH system to the user. Another reason for the use of Polish notation in FORTH is that the stack (which is basic to all FORTH operations) is a natural way to interpret a Polish expression.

Note that the non-commutative operators (subtraction and division) are evaluated in a manner such that the first number (the one preceding the operator) is the second number on the stack and the second number (the one following the operator) is the top number on the stack. This allows one to write the expression from left-to-right in the natural manner.

For example:	5 2 -	denotes	(5-2)
	9 4 /	denotes	(9/4)

EXERCISES - CHAPTER 5

- 1) Evaluate the following expression at a FORTH terminal and print the result using the `.` word (this use of a FORTH terminal could be referred to as the desk-calculator mode):

$$1 + 2*(3 + 4*(5 + 6*(7)))$$

Diagram the contents of the stack after each word is executed.

- 2) Evaluate the following expression at a FORTH terminal and print the result:

$$\frac{(1 + 2) + (3 * 4)}{(9 / 3)} - (7 * 8)$$

Diagram the contents of the stack after each word is executed.

6. THE FORTH DICTIONARY

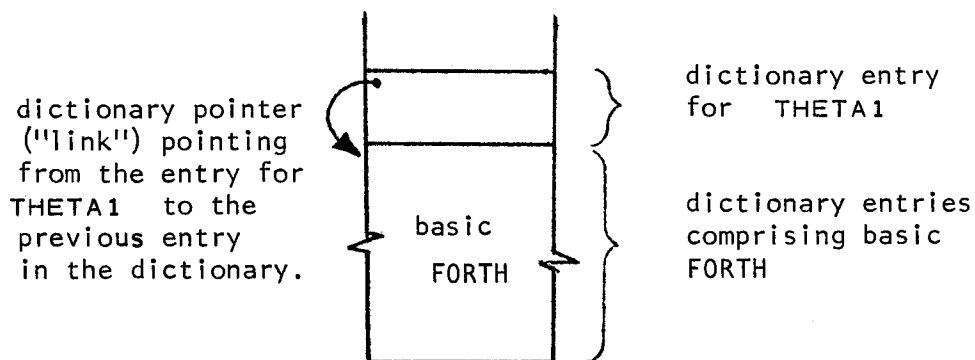
Whenever you define a word in FORTH, regardless of the purpose of the word (whether the word is a variable that contains integer values, whether the word is a sequence of instructions to execute, etc.) the word is entered into the dictionary. Loading a program into core simply consists of entering all the words defined in the program (to perform whatever functions the program is to perform) into the dictionary. The loading of basic FORTH into core (Chapter 3) is simply entering the words provided by basic FORTH into the dictionary.

Every word in the dictionary is identified by the first three characters of the word along with the count of the total number of characters in the word. Consider the following examples:

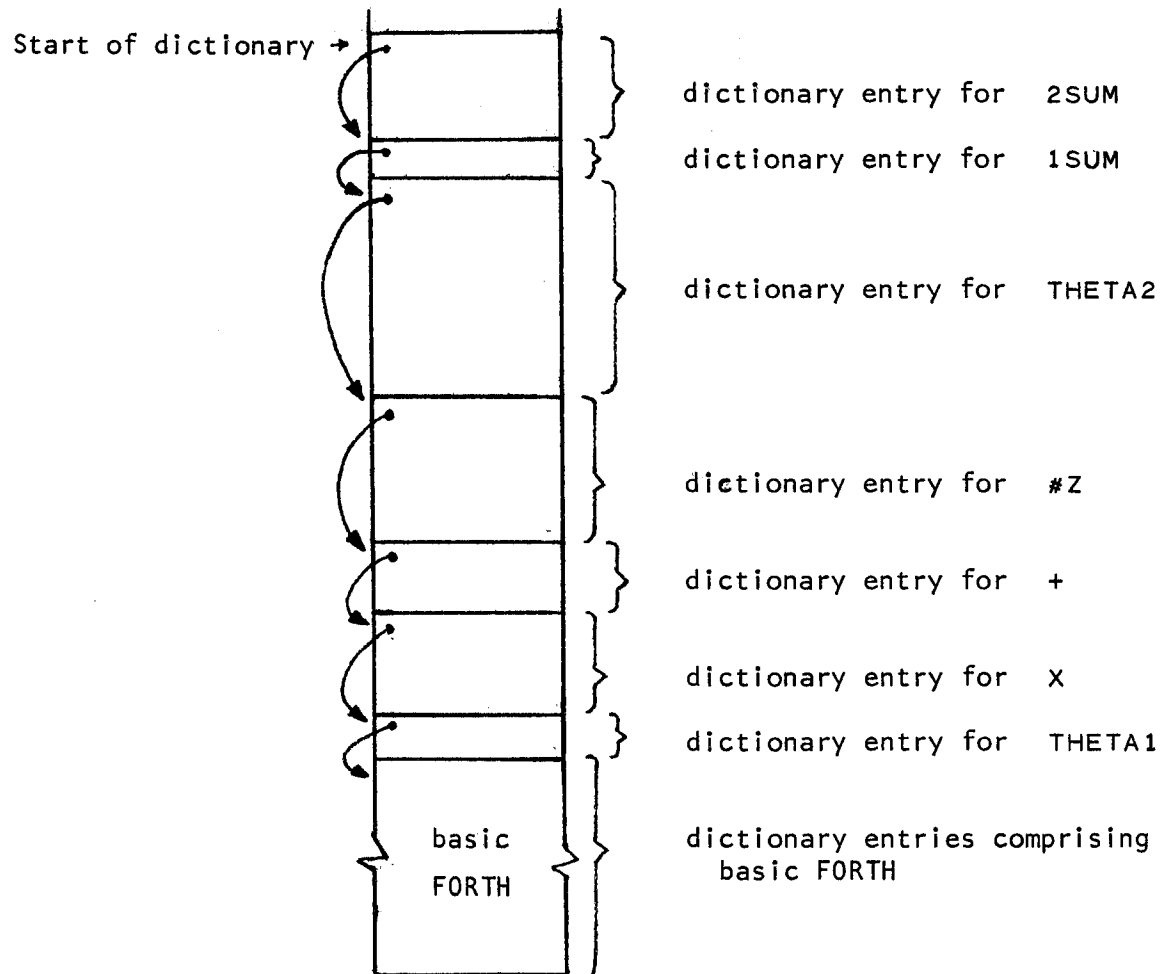
<u>WORD</u>	<u>FIRST THREE CHARACTERS</u>	<u>LENGTH</u>
THETA1	T H E	6
X	X	1
+	+	1
#Z	# Z	2
THETA2	T H E	6
1SUM	1 S U	4
2SUM	2 S U	4

Note that the words THETA1 and THETA2 have the same count and first three characters - this means these two words are indistinguishable in the dictionary. FORTH does not consider this form of redefinition an error and in fact won't even tell you about it. To avoid redefinitions of this sort you should make your variable names unique in the first three character positions (note that 1SUM and 2SUM will be distinguishable in the dictionary since the first three characters of both words are different).

There is no fixed size for an entry in the dictionary, however each entry does require a minimum of 5 words. In order for FORTH to be able to find its way through the dictionary, a dictionary chain is built with each entry pointing to the previous entry. Assume that basic FORTH has been loaded into core and the user then enters each word in the above example into the dictionary. After entering the word THETA1 the dictionary would appear as:



Similarly, if the words are entered in the sequence shown we will obtain a dictionary structure of the form:



Note that in this discussion of the dictionary we do not care what function each word performs or how the word was entered into the dictionary (whether it is a variable, a sequence of instructions, etc.) but simply in the structure of the dictionary. It is very important to note that each word is identified in the dictionary simply by its first three characters and count. When the dictionary is being searched for a specific word (say for example that you enter the word THETA5 on the terminal and FORTH must then search the dictionary to locate the entry for THETA5. in order for the word to be executed) the search starts with the last word entered and proceeds down the dictionary chain. When searching for THETA5, FORTH starts with the entry for 2SUM, then proceeds to the entry for 1SUM, then proceeds to the entry for THETA2 and this entry is a match since the first three characters and count are identical with the first three characters and count of THETA5. Hopefully this is what the user wanted! It should be obvious that after THETA2 is entered into the dictionary the entry for THETA1 is inaccessible.

EXERCISES - CHAPTER 6

- 1) Assume that the words below are entered into the dictionary in the following order:

Y+X
X+Y
XANG
X+Y2
SINX
YANG
SINY
S1NZ
XANT
X+Y/Z

- a) Which entry is executed if SINX is entered?
b) Which entry is executed if X+Y is entered?
c) Which entry is executed if XANT is entered?
d) Which entry is executed if XANG is entered?
e) Which entry is executed if SIN5 is entered?
f) Which entry is executed if Y+XX is entered?

7. DATA STRUCTURES

All of the examples in the previous chapters have dealt with numbers (not variables) and only with integer values. This chapter describes the facilities provided to define and use variables along with the different data structures provided by FORTH.

7.1 INTEGERS

An integer in Varian FORTH must be in the range

$$-32,768 \leq \text{integer} \leq 32,767$$

and occupies a single computer word. Integers are also referred to as "single-word integers" and any use of the unqualified term "integer" implies a single-word integer.

There are two declarations that are available to declare either an integer variable or an integer constant:

```
<initial-value> VARIABLE <name>
<initial-value> CONSTANT <name>
```

Using either of the above declarations causes a new dictionary entry to be created and the entry is identified by the first three characters and count of <name>. The difference in the above two declarations comes when the word (identified by <name>) is executed: when a word defined as a `CONSTANT` is executed the *value* of the constant is pushed onto the stack; when a word defined as a `VARIABLE` is executed the *address of the value* of the integer is pushed onto the stack. Once the address of the value of an integer is on the stack the value may be pushed onto the stack or a new value may be stored by using the load operator (the at-sign @) or the store operator (the exclamation-point, !). As confusing as this appears some examples should hopefully make it clear:

```
5  CONSTANT  X
180 CONSTANT  REV
```

Executing the word `X` pushes the number 5 onto the stack. Similarly, executing the word `REV` pushes the number 180 onto the stack.

```
0  VARIABLE  ZERO
-2 VARIABLE  DELTAX
```

Executing the word `ZERO` pushes the address of the integer's value onto the stack and if the word `@` is then executed the number 0 is pushed onto the stack (replacing the address). That is

```
ZERO  @      pushes 0 onto the stack,
DELTAX @      pushes -2 onto the stack.
```

In order to change the value of `ZERO` to -1 the sequence:

```
-1 ZERO ! stores -1 as the value of ZERO.
```

Similarly to change the value of `DELTAX` to +2:

```
+2 DELTAX ! stores +2 as the value of DELTAX.
```

To store a new value in a word defined as a `CONSTANT` the sequence

```
4 ' REV ! stores 4 as the value of REV.
```

Similarly,

```
-99 ' X ! stores -99 as the value of X.
```

Subsequent execution of the word `REV` would push the number 4 onto the stack and similarly executing `X` would push -99 onto the stack.

Thus we have two methods to define a word that is to contain an integer value and for each method the integer value may be pushed onto the stack or a new value may be stored as its value. This may be summarized as:

	to push current value onto stack	to store top of stack as new value
CONSTANT	<name>	' <name> !
VARIABLE	<name> @	<name> !

These two fairly similar definitions may seem somewhat superfluous since each can perform the same function, namely pushing the value of an integer variable onto the stack and storing the top number on the stack as the variable's new value. The usefulness of each will become more apparent when colon definitions are introduced, however you can still note the fact that if the value of an integer variable will be referenced (i.e. - its value pushed onto the stack) many more times than a new value is to be stored in the variable, then it requires fewer words to push the value onto the stack for a CONSTANT than for a VARIABLE. This is so because to push the value of a CONSTANT onto the stack you simply execute the word itself. To push the value of a VARIABLE onto the stack you must execute both the word and then the load operator, @.

It is worthwhile now to consider the interaction of the stack in executing a sequence of loads and stores. Consider the following four lines of input:

```

1 VARIABLE X
8 CONSTANT Y
X @ Y + ' Y !
Y .

```

WORD FROM INPUT	ACTION	CONTENTS OF STACK
1	FORTH interprets this word as a number and pushes its value onto the stack ----->	<div>1</div>
VARIABLE	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it will take the next word of input (X) and enter it into the dictionary. The top number on the stack (1) is used as the initial value for the dictionary entry for X. After popping the 1 from the stack, the stack will be empty ----->	empty
8	FORTH interprets this word as a number and pushes its value onto the stack ----->	<div>8</div>
CONSTANT	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it will take the next word of input (Y) and enter it into the dictionary. The top number on the stack (8) is used as the initial value for the dictionary entry for Y. After popping the 8 from the stack, the stack will be empty ----->	empty
X	FORTH looks this word up in the dictionary and then executes the word. When this word is executed, since it was defined as a VARIABLE, the address of the value of X is pushed onto the stack ----->	<div>address of value of X</div>

WORD FROM INPUT	ACTION	CONTENTS OF STACK
a	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it takes the the top value on the stack (which must be the address of a variable's value) and uses this address to push onto the stack the current value of the variable ----->	1
Y	FORTH looks this word up in the dictionary and then executes the word. When this word is executed, since it was defined as a CONSTANT, the value of the variable is pushed onto the stack ----->	8 1
+	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it will add together the top two numbers on the stack and then replace these two numbers with their sum -->	9
.	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it will take the next word of input (Y) and look this new word up in the dictionary. The address of the value of this new word is then pushed onto the stack ----->	address of value of Y 9
!	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it expects an address of a variables value to be on top of the stack and a number to be below on the stack. The value of the variable is set to the number and then both the address and the number are popped from the stack ----->	empty

WORD FROM INPUT	ACTION	CONTENTS OF STACK
Y	FORTH looks this word up in the dictionary and then executes the word. When this word is executed, since it was defined as a CONSTANT , the value of the variable is pushed onto the stack ----->	9
.	FORTH looks this word up in the dictionary and then executes the word. When this word is executed it prints the top number on the stack and then pops the number from the stack ----->	empty

The concepts presented in this section are fundamental to understanding the remainder of the chapter and the reader is advised to work the first two exercises at the end of the chapter before proceeding on.

7.2 DOUBLE-WORD INTEGERS

There are occasions when the single-word integer (described in the previous section) does not contain enough precision for a certain application. In these cases a double-word integer may be used.

In Varian FORTH the range is

$$-1,073,741,824 \leq \text{double-word integer} \leq 1,073,741,823$$

and each double-word integer occupies two consecutive words of memory and will occupy two words on the stack.

There are two words that are available to declare either a double-word integer variable or a double-word integer constant:

```
<initial-value> 2VARIABLE <name>
<initial-value> 2CONSTANT <name>
```


Using either of the above declarations causes a new dictionary entry to be created and the entry is identified by the first three characters and count of <name>. The difference in the above declarations comes when the word (identified by <name>) is executed - when a word defined as a `2CONSTANT` is executed the *2-word value* of the double-word constant is pushed onto the top two words of the stack; when a word defined as a `2VARIABLE` is executed the *address of the value* of the double-word integer is pushed onto the stack. Once the address of the value of a double-word integer is on the stack the 2-word value may be pushed onto the top two words of the stack by using the double-word load operator (`D@`). Similarly the double-word store operator (`D!`) may be used to store a new 2-word value in a double-word integer.

Before giving examples of the above declarations it must be noted that a number must be appended by a *comma with no numbers appearing to the right of the comma* if the number is to be interpreted by FORTH as a double-word integer value.

For example, the sequence

```
0 ,  
25 ,  
100000 ,
```

correctly enters three double-word integers whose values are 0, 25 and 100,000. The following is an error

```
DPREC  
10,0
```

Since the digit zero appears to the right of the comma, the number is not interpreted as 10 but rather as 100! This required usage of the comma probably seems unnecessarily complex, however, FORTH must have a way of uniquely separating single-word integers from double-word integers and the comma convention is the method used.

Consider the declaration

```
99, 2VARIABLE XX
```

which defines a double-word integer variable named `XX`, with an initial value of 99. The sequence

```
XX D@
```

pushes the current 2-word value of `XX` onto the top two words of the stack. The sequence

```
500001, XX D!
```

stores 500001 as the new value of `XX`. Now consider the declaration

```
1, 2CONSTANT D1
```

which defines a double-word integer constant named `D1`, with an initial value of 1. The sequence

```
D1
```

pushes the 2-word value of `D1` onto the top two words of the stack. To store 100 as the new value of `D1` the sequence

```
100, ' D1 D!
```

is required. Summarizing the results of the previous section and the results of this sections yields the table:

	to push current value onto stack	to store top value on stack as variable's new value	
VARIABLE	<name> @	<name> !	value occupies one word on stack
CONSTANT	<name>	' <name> !	
2VARIABLE	<name> D@	<name> D!	value occupies two words on stack
2 CONSTANT	<name>	' <name> D!	

Now consider the interaction of the stack in executing a sequence of double-word loads and stores, using the variables `XX` and `D1` declared above. Assume the line of input is as follows:

```
XX D@ D1 D+ ' D1 D!
D1 D.
```

(The word `D+` is the double-word addition operator, which adds together two double-word integer values, similar to the single-word addition operator `+`. We also have the double-word subtraction operator `D-`).

WORD FROM INPUT	ACTION	CONTENTS OF STACK
XX	Push address of 2-word value onto stack ----->	<div>address of value of XX</div>
D@	Take address on top of stack and replace it with 2-word value ----->	<div>99</div>
D1	Push the 2-word value onto stack ----->	<div>1</div> <div>99</div>
D+	Add together the two double-word integer values on top of the stack and then push the result onto the stack ----->	<div>100</div>
'	Take the next word of input (D1) and push the address of its value onto the stack ----->	<div>address of value of D1</div> <div>100</div>
D!	Take the address on top of the stack and store the 2-word integer value beneath the address at the address, then pop the address and the 2-word value off the stack ----->	empty
D1	Push the 2-word value onto the stack ----->	<div>100</div>
D.	Print the 2-word integer value on top of the stack, then pop the 2-word value off the stack ----->	empty

The words in FORTH to operate on single-word integers (the words VARIABLE CONSTANT + - * / . etc.) are all defined within basic FORTH and therefore will be in the dictionary after the user loads basic FORTH (Chapter 3). These words are considered essential to any FORTH program and are therefore always available. The words used to operate on double-word integers (the words 2VARIABLE 2CONSTANT D+ D- D. etc.) are considered optional and are not automatically entered into the dictionary with basic FORTH. Instead, if you wish to use double-word integers you must specifically load the appropriate words into the dictionary by executing the word USER.

This will load into core (i.e. - enter into the dictionary) the double-word integer routines, the floating-point routines and other miscellaneous routines to be described later.

7.3 FLOATING-POINT NUMBERS

The data structures presented in the previous sections (single-word integer and double-word integer) are *exact* representations of integer values. This means that arithmetic performed on single-word integers or double-word integers yields exact answers: $2 * 3$ is exactly 6, not 5.999... ; $5 + 9$ is exactly 14, not 14.000001. As long as the user stays within the range of the data structure (which for a double-word integer is -1,073,741,824 to +1,073,741,823) this guarantee of exactness for integer arithmetic holds. The obvious drawback in using integers is the limited range that is available - in Varian FORTH a double-word integer corresponds to just over 9 digits of precision. For some applications (notably financial programming with the COBOL language) the exactness of integer arithmetic is a requisite *and* the data structures being represented will never exceed approximately 15-20 digits, hence exact arithmetic is used (using a slight variation of integer representation).

Scientific programming, on the other hand, requires a data structure that provides a far greater range of values so that very large numbers (for example, Avagadro's number $N = 6.02250 \times 10^{23}$) and very small numbers (for example, Planck's constant $h = 1.0545 \times 10^{-27}$) can both be stored. Additionally, scientific programming does not require that these numbers be represented *exactly* but only that a specified number of digits of accuracy be maintained. The data structure provided by most programming languages (and by FORTH) for scientific computation is floating-point. A floating-point number is stored in the computer as a fraction times an exponent, similar to standard scientific notation. In Varian FORTH a floating-point number is stored in three consecutive words of memory and each floating-point number requires three words on the stack. The fraction will be in the range

$$0.0 \leq |\text{fraction}| < 1.0$$

and provides approximately 9 digits of accuracy. The exponent will be in the range

$$-32,768 \leq \text{exponent} < 32,767$$

and denotes the power of 2 that the fraction is raised to, that is

$$\text{floating-point number} = \text{fraction} * 2^{\text{exponent}}$$

In using floating-point numbers one is sacrificing exactness for greater range. (If one were to store both Avagadro's number and Plank's constant exactly, a computer word of approximately 100 bits would be required!)

There are two words that are available to declare either a floating-point variable or a floating-point constant:

<initial-value> REAL <name>

<initial-value> FCONSTANT <name>

Using either of the above declarations causes a new dictionary entry to be created and the entry is identified by the first three characters and count of <name>. The difference in the above two declarations comes when the word (identified by <name>) is executed - when a word defined as an FCONSTANT is executed the *3-word value* of the floating-point constant is pushed onto the stack; when a word defined as a REAL is executed the *address of the 3-word value* of the floating-point variable is pushed onto the stack. Once the address of the 3-word value of a floating-point variable is on the stack the floating-point load operator (F@) or the floating-point store operator (F!) may be used.

A floating-point number *must* contain a decimal point and may optionally contain digits to the right of the decimal point.

Consider the following examples:

3.14159	→	.314159 * 10 ¹	} floating-point numbers
200.1	→	.2001 * 10 ³	
1.000	→	.1 * 10 ¹	
.27	→	.27 * 10 ⁰	
1,	→	1	} double-word integers
1,0	→	10	
1,50	→	150	
1.0000	→	.1 * 10 ¹	} floating-point number

Consider the declaration

```
180.0 REAL THETA
```

which defines a floating-point variable named THETA, with an initial value of 180.0. The sequence

```
THETA F@
```


pushes the current 3-word value of THETA onto the top three words of the stack. The sequence

```
90.000 THETA F!
```

stores 90. as the new value of THETA . Now consider the declaration

```
3.14 FCONSTANT PI
```

which defines a floating point constant named PI , with an initial value of 3.14. The sequence

```
PI
```

pushes the 3-word value of PI onto the top three words of the stack. To store 3.14159265 as the new value of PI (note that this new value uses the full 9-digits of accuracy provided by FORTH for a floating-point number) the sequence

```
3.14159265 ' PI F!
```

is required. Summarizing the results of the previous two sections and the results of this section yields the table:

	to push current value onto stack	to store top value on stack as variable's new value
VARIABLE	<name> @	<name> !
CONSTANT	<name>	' <name> !
2VARIABLE	<name> D@	<name> D!
2CONSTANT	<name>	' <name> D!
REAL	<name> F@	<name> F!
FCONSTANT	<name>	' <name> F!

Now consider the input line

```
PI  THETA F@ F*  THETA F!
    THETA F@  F.
```

and the interaction of the stack. (The word `F*` is the floating-point multiplication operator, similar to the word `*`. We also have the words `F+` `F-` and `F/` for floating-point addition, subtraction and division. The word `F.` prints the floating-point number on top of the stack.)

WORD FROM INPUT	ACTION	CONTENTS OF STACK
PI	Push the 3-word value onto the stack	<div> <div>3.14</div> </div>
THETA	Push the address of the 3-word value onto the stack	<div> <div>address of value of THETA</div> <div>3.14</div> </div>
F@	Take the address on top of the stack and replace it with the 3-word value	<div> <div>180.0</div> <div>3.14</div> </div>

WORD FROM INPUT	ACTION	CONTENTS OF STACK
F*	Multiply together the top two floating-point numbers on the stack and then push the result onto the stack	<div> <div>565.2</div> </div>
THETA	Push the address of the 3-word value onto the stack	<div> <div>address of value of THETA</div> <div>565.2</div> </div>
F!	Take the address on top of the stack and store the 3-word value beneath the address at the address, then pop the address and the 3-word value off the stack	empty
THETA	Push the address of the 3-word value onto the stack	<div>address of value of THETA</div>
F@	Take the address on top of the stack and replace it with the 3-word value	<div>565.2</div>
F.	Print the floating-point number on top of the stack, then pop the 3-word value off the stack	empty

Floating-point numbers may optionally be entered as a fraction times a power of ten (similar to the Fortran E format). The number 0.25 could be entered as any of the following:

```
0.25
0.25E0
25.E-2
2500.0E-4
.0000250E4
```

7.4 CONVERSIONS BETWEEN DATA STRUCTURES

Frequently the need arises to convert a number that is on the stack to another data type. For example, you may want to convert a single-word integer to a floating-point number, a floating-point number to a double-word integer and so on. The following table summarizes the words available to perform this conversion:

name of word	converts		notes
	from	-----> to	
SFLOAT	single-word integer	floating-point	
DFLOAT	double-word integer	floating-point	
SFIX	floating-point	single-word integer	(truncates)
DFIX	floating-point	double-word integer	(truncates)

Consider the following examples:

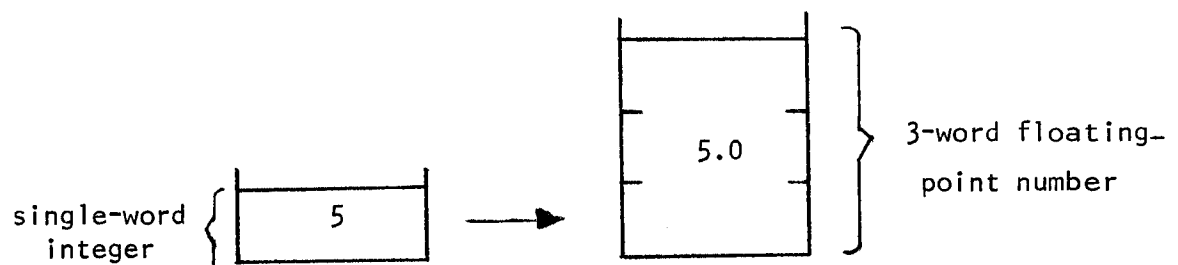
5 SFLOAT F. will print 5.0

765. DFLOAT F. will print 765.0

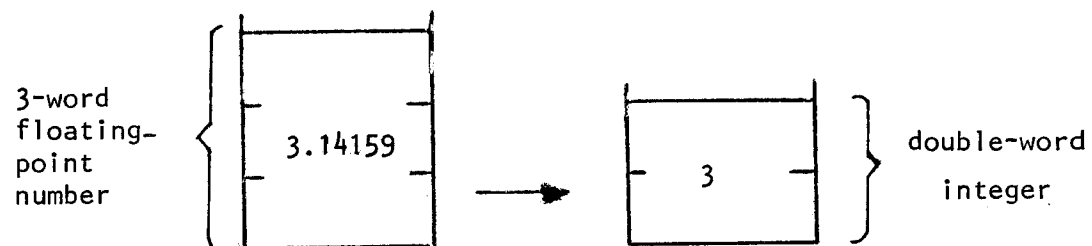
3.14159 SFIX . will print 3

1.999 DFIX D. will print 1

It is very important to remember that every conversion involves a change in the number of words of the stack used by the number. SFLOAT converts a single-word integer to a 3-word floating-point number:



Similarly DFIX converts a 3-word floating-point number to a double-word integer:



There are no specific words provided to convert a single-word integer to a double-word integer or vice-versa and since these two conversions are somewhat tricky (requiring a knowledge of the internal storage representation of the data types) their discussion is deferred until later.

7.5 LOGICAL VALUES AND LOGICAL EXPRESSIONS

There is no specific data type named logical in FORTH, instead one uses a single-word integer as a logical value and interprets the value as follows:

zero valued integer ---> FALSE

non-zero valued integer ---> TRUE

Thus whenever FORTH calls for a <logical-condition> one must provide a single-word integer value. The use of these logical conditions should become clear in the ensuing chapters.

Additionally, one may combine more than one <logical-condition> to form a logical expression using the following words:

AND Forms the logical-AND of the two single-word integer values (i.e. - logical values) on top of the stack. The logical-AND is defined as follows:

true	true	AND	--->	true
true	false	AND	--->	false
false	true	AND	--->	false
false	false	AND	--->	false

(the result is true if *both* of the operands are true).

OR Forms the inclusive-OR of the two single-word integer values (i.e. - logical values) on top of the stack. The inclusive-OR is defined as follows:

true	true	OR	--->	true
true	false	OR	--->	true
false	true	OR	--->	true
false	false	OR	--->	false

(the result is true if *either* of the operands is true).

XOR Forms the exclusive-OR of the two single-word integer values (i.e. - logical values) on top of the stack.

The exclusive-OR is defined as follows:

true	true	XOR	--->	false
true	false	XOR	--->	true
false	true	XOR	--->	true
false	false	XOR	--->	false

(the result is true if *only one* of the operands is true).

7.6 ADDITIONAL NUMERIC CONVERSIONS

All of the examples in this primer have presented the numbers in base ten, decimal. Any base may be used for numeric input and the VARIABLE BASE specifies the current base to be used for number conversion. Three words are predefined, DECIMAL OCTAL and HEX which set base to 10, 8 and 16 respectively. Furthermore, if the current base is less than or equal to ten (decimal or octal) and the last character of a number is a B then the number is converted as an octal number. For example,

20B equals 16_{10}

Since B is a legal hexadecimal digit, if the base is greater than ten a trailing B will not force octal conversion. If one wishes to force octal conversion of a double-word integer then the B must follow the comma:

20,B

is a double-word integer 16_{10} .

One of FORTH's truly unique features is its acceptance of sexagesimal (base sixty) input. This facilitates the input of angles (in degrees, minutes, seconds - such as 8:58:05) and times (in hours, minutes, seconds - such as 9:01:09). FORTH accomplishes this by allowing a colon to appear within any number and then converting the digit immediately *following* the colon using base 6.

All digits following this first digit

after the colon are converted according to the current base (i.e. - decimal or octal, as explained above). For example, the

number 000100 is interpreted as 100, while the number

00:01:00 is interpreted as 60. This latter number could represent, for example, the time-of-day of one minute past midnight which equals 60 seconds past midnight. Similarly, 00:03:00 is interpreted

as 180. Hence, if the quantity being entered is a time, then the resulting value will represent the number of seconds past midnight.

If the quantity is an angle, then the resulting value represents the number of seconds of angle. Additionally, if a floating-point number is used then a time may be specified as fractions of a second or an angle may be specified as fractions of a second. For example:

12:00:01.5 could represent the time of one and
 a half seconds past noon; also it
 could represent the angle 12° 00' 01.5".

Note that since the digit following the colon is interpreted as base 6, this digit can only be 0 through 5. The number 6:80, for example, is illegal. This also means that the minutes and seconds of either an angle or a time must be entered as *exactly* two digits. The time of one minute past eight must be entered as either

8:01:00 or 08:01:00

and not as

8:001:00 or 8:1:00

The use of sexagesimal rotation in a number forces the number to be interpreted as a double-word integer unless the number contains a decimal point. For example

01:00	→	double-word integer	60
01:00,	→	double-word integer	60
01:00.	→	floating-point	60.0

7.7 VECTORS

FORTH provides the ability to declare a vector of either integers, double-word integers or floating-point numbers. This data structure is similar to a one-dimensional array in Fortran. FORTH does not provide any multi-dimensional arrays. (However, given the structure of FORTH one could define new words to declare and access multi-dimensional arrays. Unfortunately, this is beyond the scope of this primer.)

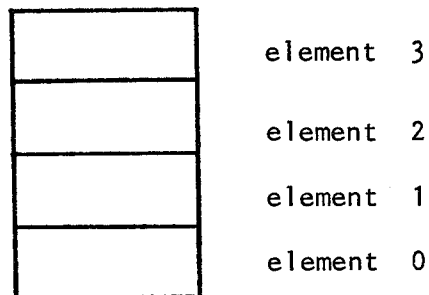
To declare a vector of single-word integers one would execute

```
<maximum-subscript>  ()DIM  <name>
```

For example, the following defines a four element vector named X :

```
3  ()DIM  X
```

The four elements would be accessed by using the subscripts 0, 1, 2 and 3 and a memory diagram might be



Note that the subscript starts at 0, therefore the number of elements equals <maximum-subscript> + 1.

In order to access a single element one executes

```
<subscript-value>  <name>
```

and, for example, the sequence

```
3  X
```

calculates the address of element 3. This address may be used just like the address of any integer value, for example

1 X @

pushes the value of element 1 onto the stack. The sequence

0 X @ 3 X !

takes the value of element 0 and puts it in element 3. Finally the sequence

0 X @ 1 X @ + 2 X @ + 3 X @ +

forms the sum of all elements in the vector X.

Similarly we may declare a vector of double-word integers

<maximum-subscript> 2()DIM <name>

and a vector of floating-point numbers

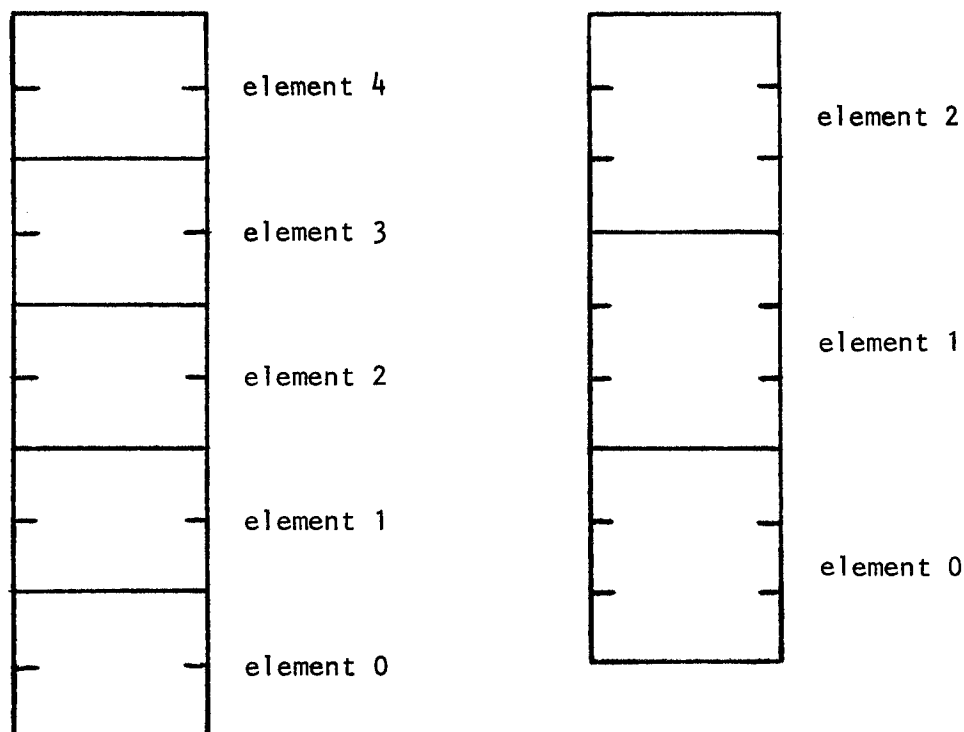
<maximum-subscript> 3()DIM <name>

The two sequences

4 2()DIM Y

2 3()DIM Z

would declare a 5 element vector of double-word integers and a 3 element vector of floating-point numbers. Their memory representation could be



To obtain the address of a specific element one executes

<subscript-value> <name>

For example

4 Y D@ DFLOAT 2 Z F!

will push element 4 of the vector Y onto the stack, convert it from a double-word integer to a floating-point number and store this floating-point value in element 2 of Z. Naturally there is no requirement that the <subscript-value> be a numeric constant and the following is perfectly valid:

0 VARIABLE INDEX
2 INDEX ! INDEX @ X @

EXERCISES - CHAPTER 7

- 1) Define a VARIABLE I (with initial value 5) and a CONSTANT J (with initial value 100) and calculate the following series of expressions:

```
I = I + 1
J = I * 5
PRINT I, J
I = (J * J) - 10
J = (I / J) + 2
PRINT I, J
```

- 2) Define a VARIABLE A (with initial value 20) and a CONSTANT B (with initial value 32) then calculate the following expression:

$$B = A + \frac{B}{A + \frac{B}{A - 1} + \frac{A}{B}}$$

Then print the value that was stored in B.

- 3) Define a VARIABLE II (with initial value 10) and a DCONSTANT JJ (with initial value 30) and calculate the following expressions:

```
II = JJ + 1
JJ = JJ - II
PRINT II, JJ
II = II + JJ + 101
JJ = JJ + II
PRINT II, JJ
```

4) Code the expression

$$A = B * (C + D)$$

in two ways, using the fact that addition is a distributive operator (i.e. $- b * (c + d) = b * c + b * d$).

Assume that

A	is a	CONSTANT,	initial value	1
B	is a	VARIABLE,	initial value	5
C	is a	CONSTANT,	initial value	8
D	is a	VARIABLE,	initial value	25.

Confirm that both methods generate the same answer.

5) Code the expression

$$A = B + C + D + E$$

in two ways, using the fact that addition is a commutative operator (i.e. $- b + c + d + e = e + d + c + b$). Assume that

A	is a	2CONSTANT,	initial value	3
B	is a	2 VARIABLE	initial value	5
C	is a	2CONSTANT,	initial value	7
D	is a	2 VARIABLE	initial value	11
E	is a	2CONSTANT,	initial value	17.

Confirm that both methods generate the same result.

6) What expression do the following lines of FORTH code correspond to?

```
7.0 REAL A
8.0 REAL B
1.0 REAL C
```

```
B F@ B F@ F* 4.0 A F@ F* C F@ F* F-
```

What value is obtained when the expression is evaluated?

Where is this value stored?

7) What is the largest time-of-day (counted in seconds) that may be stored in a single-word integer? In a double-word integer?

8) Using the definitions

```
5 VARIABLE I
12 CONSTANT J
```

```
280. 2VARIABLE A
257. 2CONSTANT B
```

```
1.325 REAL X
5.0 FCONSTANT Y
```

Calculate and print the value of the expression

$$Y = \frac{(I * X)}{(A - B)} / (J/Y)$$

9) Will the following FORTH expressions be interpreted as true or false?

- a) 1 2 + 4 /
- b) 1 0 AND
- c) 1 1 AND
- d) 0 0 OR
- e) 5 8 / 2 1 - XOR
- f) 1 2 2 - OR

10) What will the following FORTH expression print?

```
DECIMAL 10 OCTAL 10 - DECIMAL .
```


8. STACK OPERATIONS

All of the stack operations up to this point have involved pushing a number onto the top of the stack, popping a number off the top of the stack and performing arithmetic on the top two numbers on the stack. This chapter describes some additional operations that may be performed on the stack.

8.1 MANIPULATION WORDS

Frequently one can "optimize" the evaluation of an arithmetic expression by intelligently using the stack to hold temporary results, rather than storing every temporary result in a variable. For example consider the declarations:

```
4 VARIABLE A
18 VARIABLE B
0 VARIABLE X
0 VARIABLE Y
```

and the subsequent evaluation of the expressions

```
X = A + B
Y = X + 1
```

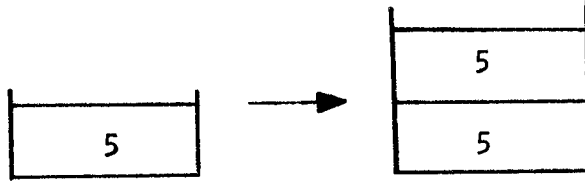
Using the methods described in this manual up to this point one would code this in FORTH as

```
A @ B @ + X !
X @ 1 + Y !
```

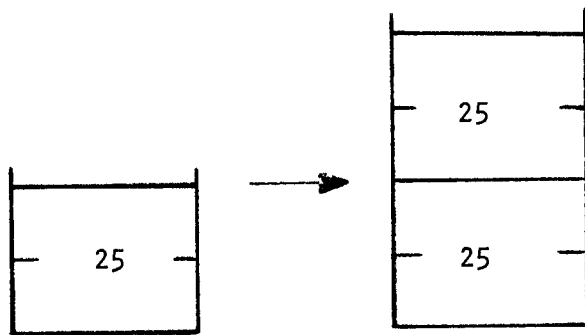
which is perfectly correct and acceptable. Now assume we have available to use the word `DUP` which simply duplicates the single-word quantity on top of the stack. For example

```
5 DUP
```

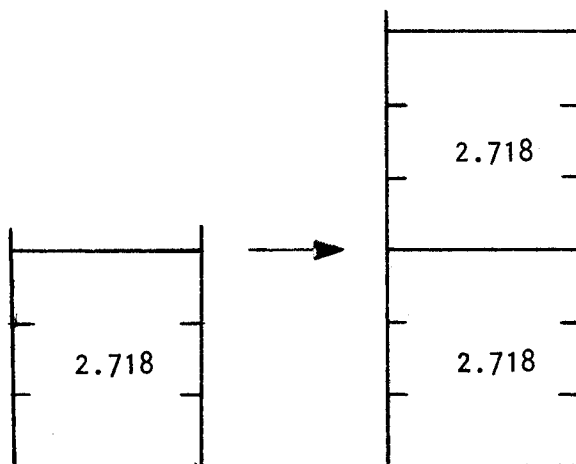
performs the following stack operations:



Similarly we have 2DUP and 3DUP which duplicate 2-word stack entities and 3-word stack entities:

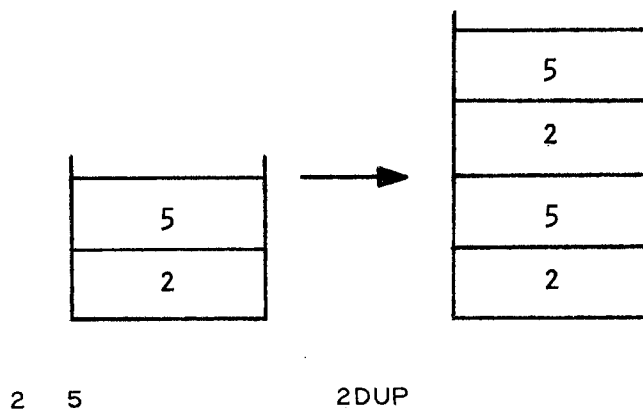


25 , 2DUP



2.718 3DUP

Note that the terms "single-word quantity", "2-word entity" and "3-word entity" are used instead of "single-word integer", "double-word integer" and "floating-point number" to describe the stack entries being manipulated. This is because unlike the arithmetic operators, which expect the top entries on the stack to be a specific type of number, the stack manipulation words being described in this chapter do not care what internal form of data they are working on. The word `DUP` will gladly duplicate a single-word integer or a 1-word address. Similarly the word `2DUP` could be used to duplicate either a double-word integer or two single-word integers, as follows:



The stack is a general purpose "tool" and as such a variety of different entities may reside on the stack for totally different purposes. It is important to always know, when writing a program, what quantities are on the stack and what format these quantities are in (number, address, etc.). FORTH will gladly allow you to perform erroneous operations on the wrong type of data structure (such as taking the square root of a single-word address, thinking the address was really a single-word integer) although the results are usually disastrous (and sometimes hard to find).

Getting back to the original example we may alternately code

```

X = A + B
Y = X + 1

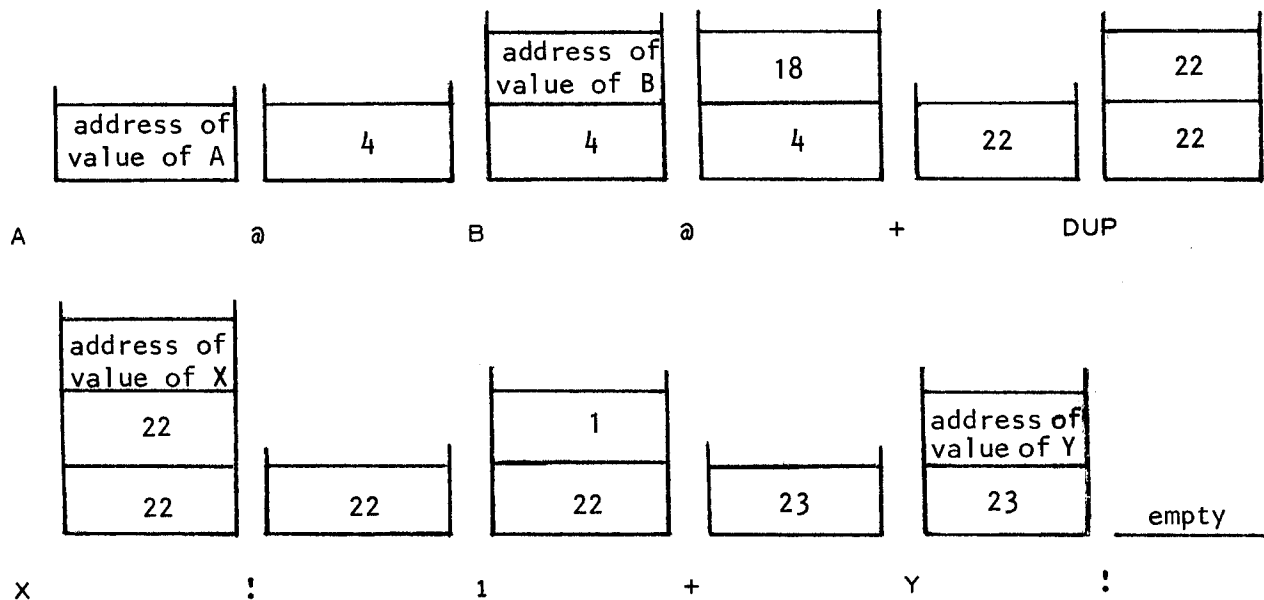
```

as

```
A @ B @ + DUP X !
1 + Y !
```

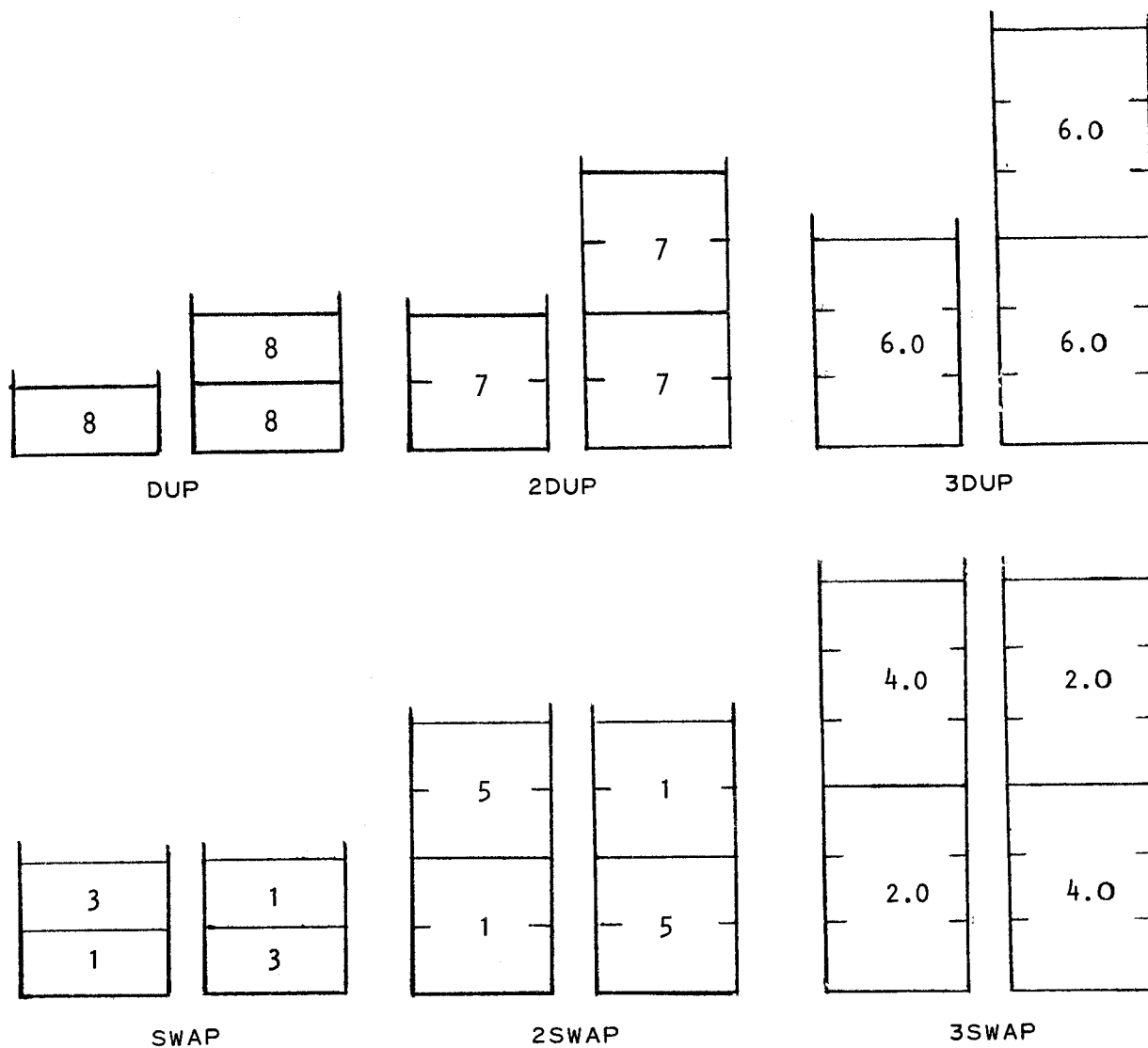
What we have done is duplicate the quantity being stored in X (namely A + B) and then after storing one copy of this result in X we use the second copy to compute $Y = X + 1$. This saves having to load X from core back onto the stack in order to calculate $Y = X + 1$.

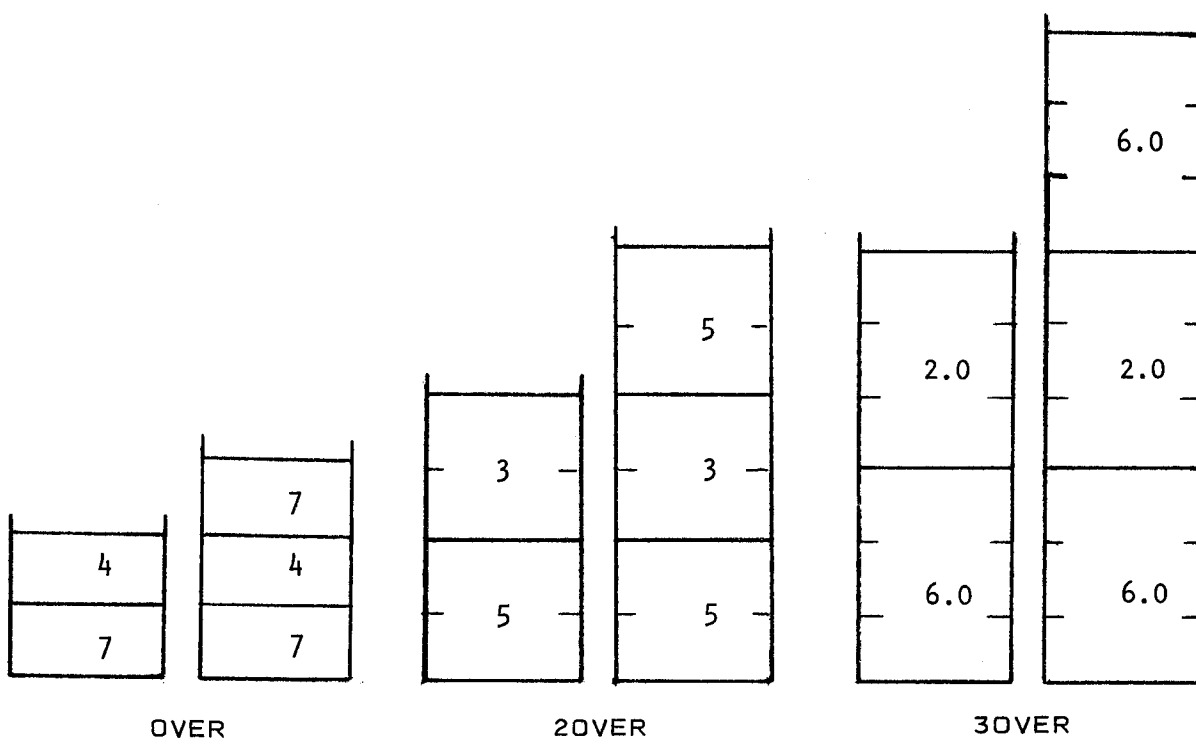
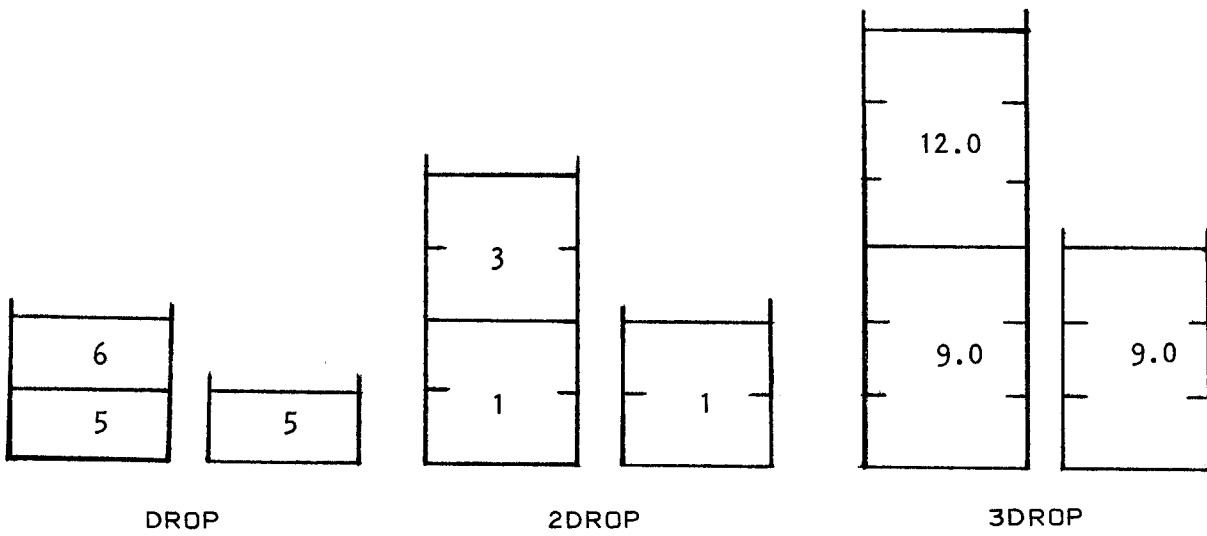
The sequence of stack operations is as follows:

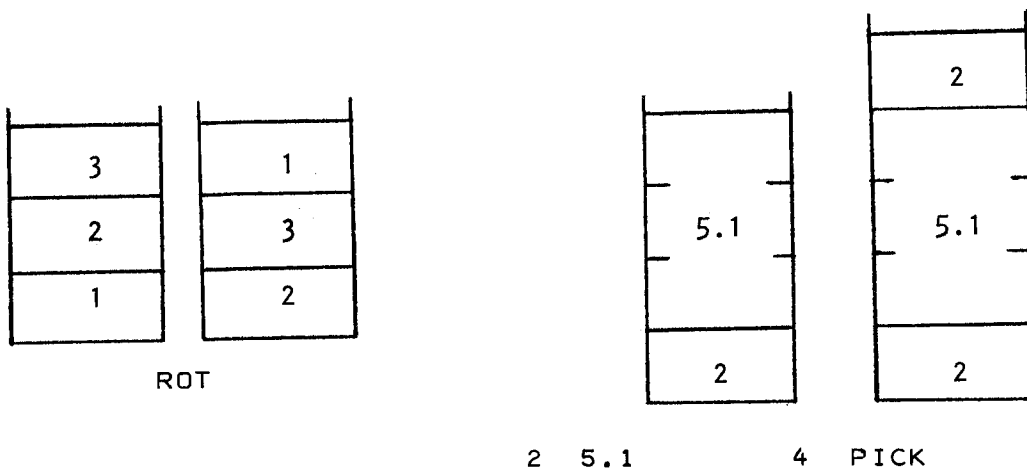


Although this example may seem somewhat trivial (so what if we save a single load?) on some computers stack manipulations may be *many* times faster than core-to-stack or stack-to-core operations so that it can really be beneficial to retain values on the stack whenever possible. One must not go overboard and try to keep everything on the stack at all times or you will soon lose track of just what is on the stack.

A summary of the available stack operations is provided in table 8.1 and examples of the important operators are given below. All examples will be given using numbers as the stack entries although as mentioned previously, the use of these operators is not restricted to numbers.







One error that will be detected by FORTH is stack underflow. This occurs when the stack is empty and the program attempts to operate on the stack. (The only valid operation to perform on an empty stack is to push a number onto the stack.) For example, the sequence

1 +

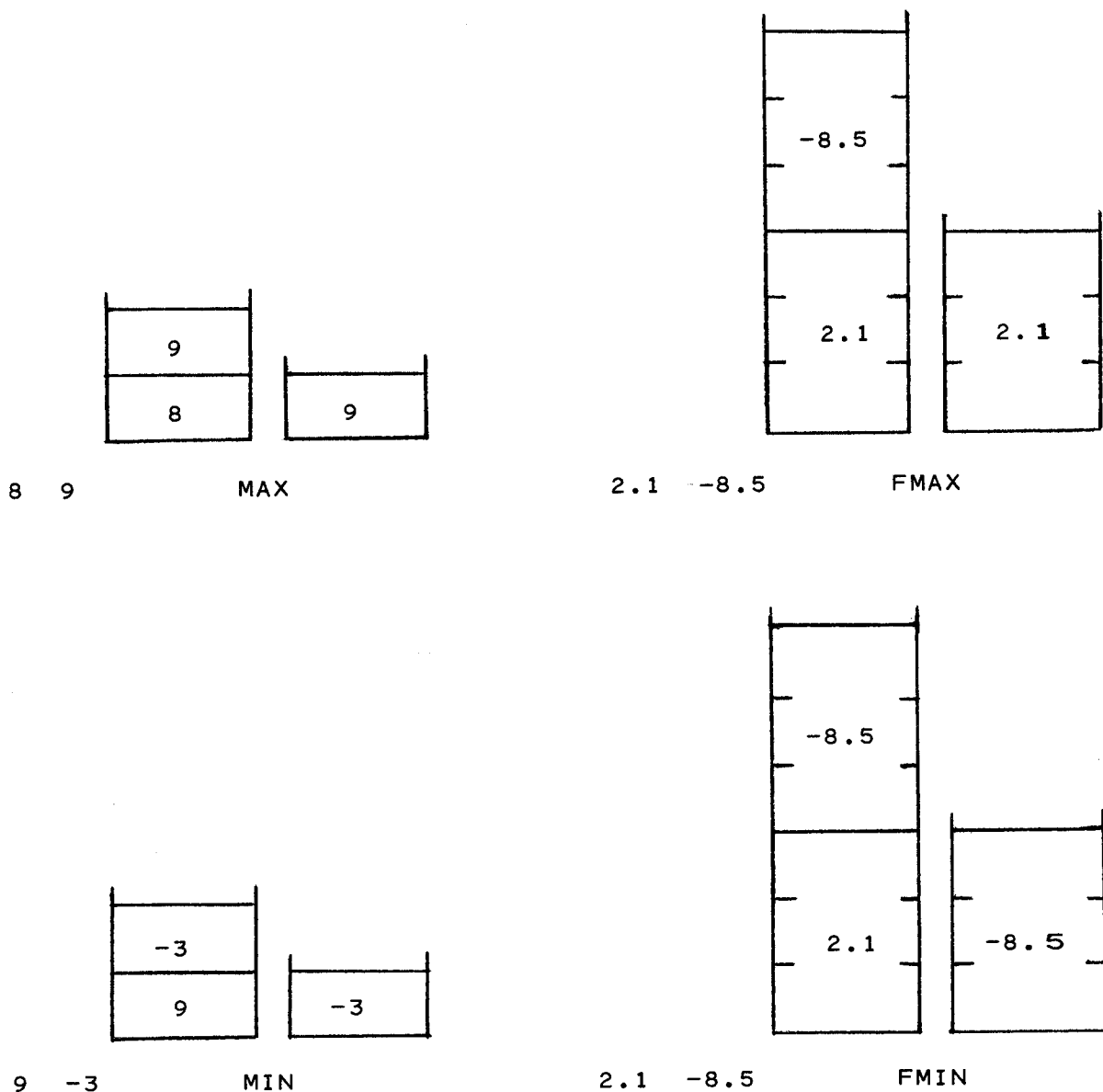
will generate the response + ?U since the single-word addition operator expects *two* single-word integers on the stack. (This example assumes the stack was empty prior to entering the sequence 1 +).

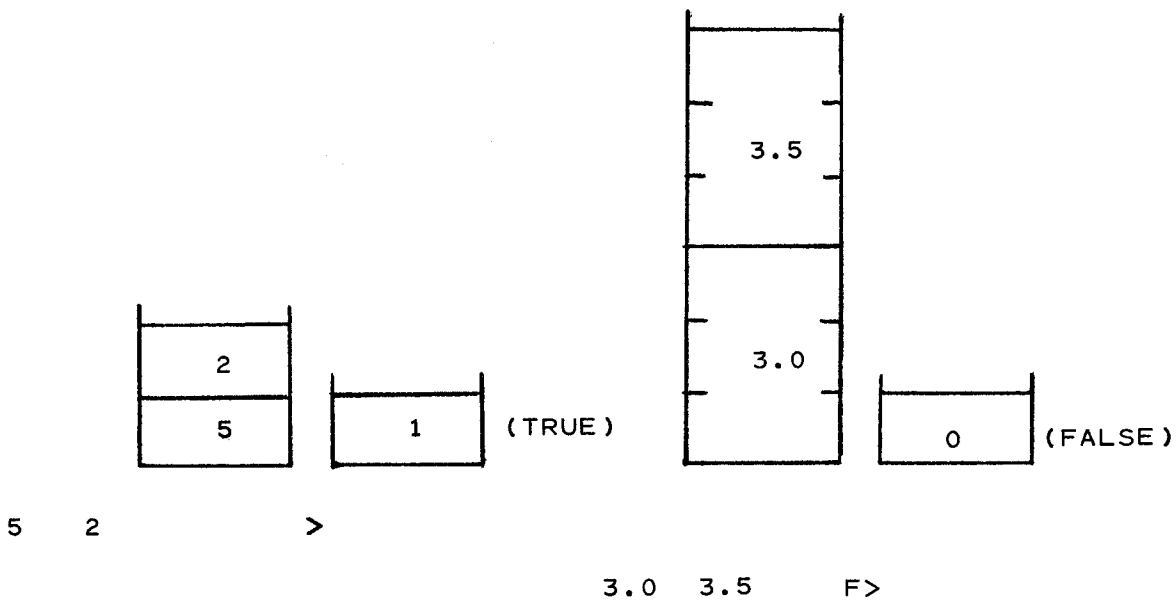
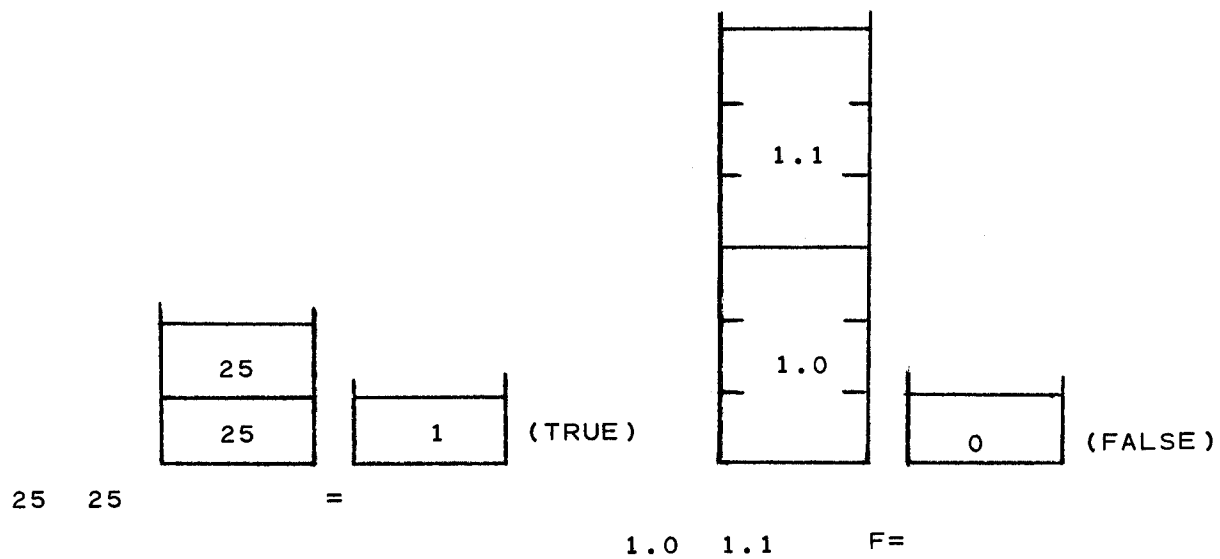
1-Word Operators	2-Word Operators	3-Word Operators	Description
DUP	2DUP	3DUP	Duplicates the top entry on the stack.
SWAP	2SWAP	3SWAP	Interchanges the top two entries on the stack.
DROP	2DROP	3DROP	Deletes the top entry from the stack.
OVER	2OVER	3OVER	Takes a copy of the second entry on the stack and pushes this copy onto the top of the stack.
ROT	2ROT	3ROT	Rotates the top three words on the stack, moving the third to the top, the second to the third and the top to the second.
PICK	2PICK	3PICK	Takes a copy of the n^{th} word on the stack and pushes this copy onto the top of the stack, where n is the top word on the stack. 1 PICK is identical to DUP and 2 PICK is identical to OVER.
AND			Replaces the top two words with their logical AND.
OR			Replaces the top two words with their inclusive OR.
XOR			Replaces the top two words with their exclusive OR.

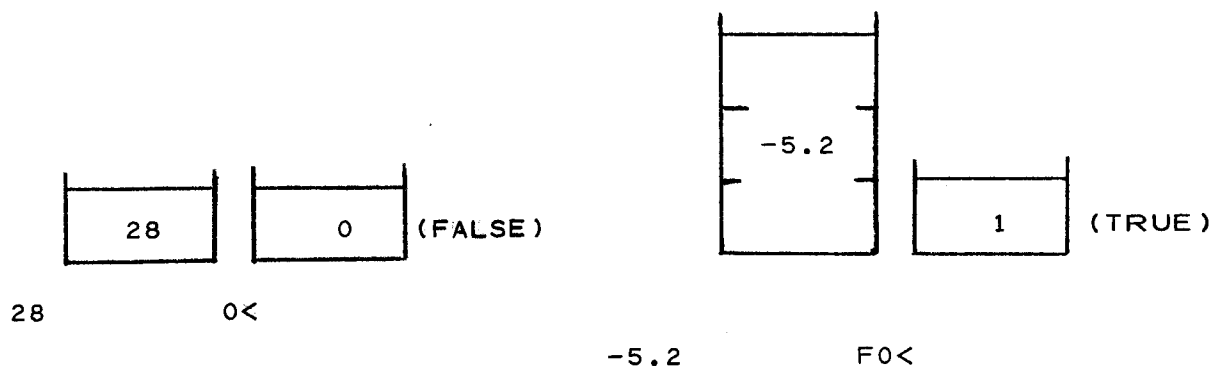
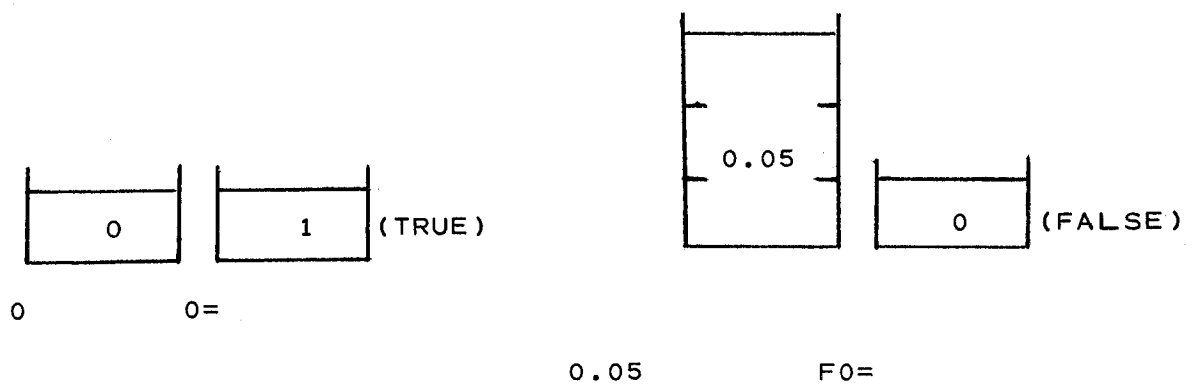
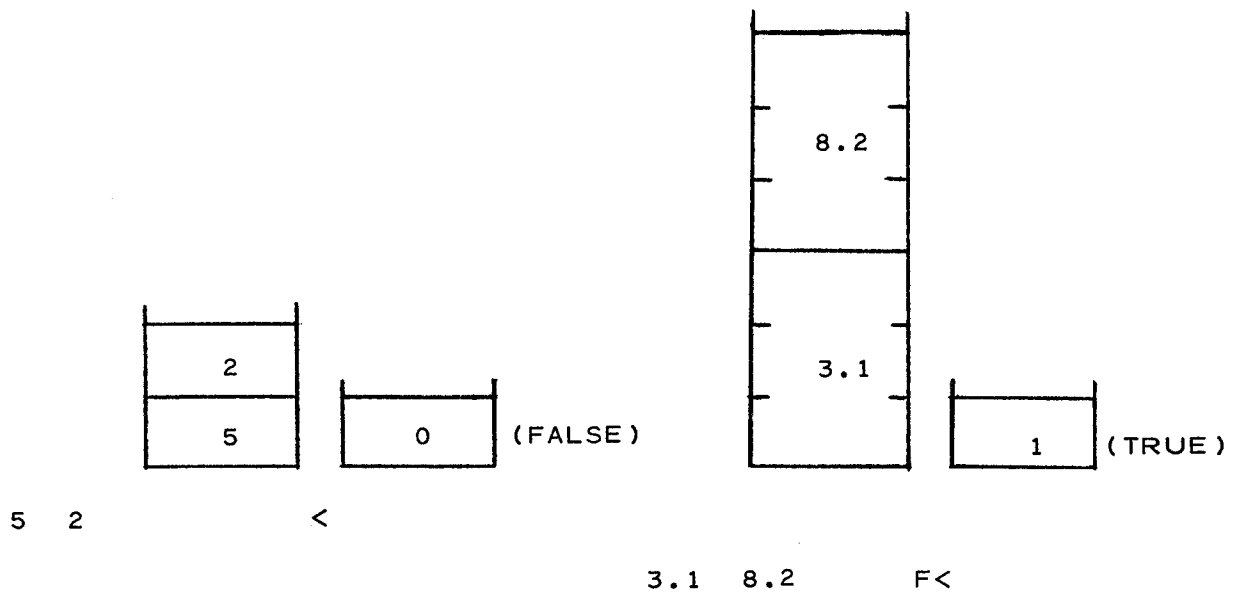
TABLE 8.1 - Manipulation Words

8.2 COMPARISON WORDS

These words numerically compare the top one or two numbers on the stack leaving a logical value (true or false, as described in Section 7.5) as the result. These words operate on either single-word integers or floating-point numbers as shown in Table 8.2. Examples of these words are given below.







Single-word Integer	Double-word Integer	Floating Point	Description
MAX	DMAX	FMAX	Replaces the top two numbers with the number having the greater magnitude.
MIN	DMIN	FMIN	Replaces the top two numbers with the number having the lesser magnitude.
= < > <> >= <=	D= D< D>	F= F< F>	Replaces the top two numbers with the logical value TRUE (non-zero) or FALSE (zero) depending on their relationship.
O= O< O> O<> O>= O<=	D0= D0<	F0= F0<	Replaces the top number with the logical value TRUE (non-zero) or FALSE (zero) depending on the relationship between the number and zero.

TABLE 8.2 - Comparison Words

EXERCISES - CHAPTER 8

1) Given the definitions

```
5. 2VARIABLE  X
9.  2CONSTANT Y
0.  2CONSTANT Z
```

evaluate the expression

$$Z = 2 * (X - Y)$$

in two ways. Print the result that is stored in Z. Note that there is not a double-word integer multiply.

2) Using the definitions in question 1, evaluate the expression

$$Z = (X - Y) + (X + Y)$$

loading the value of X onto the stack only once. Print the result that is stored in Z.

3) Using the definitions in question 1, evaluate the expression

$$Z = (X + Y) + (X - Y)$$

loading the value of X onto the stack only once and loading the value of Y onto the stack only once. Print the result that is stored in Z.

4) What is the final contents of the stack after each of the following sequences is executed?

a) 5 DUP 4 + -

b) 12 3 OVER SWAP DROP +

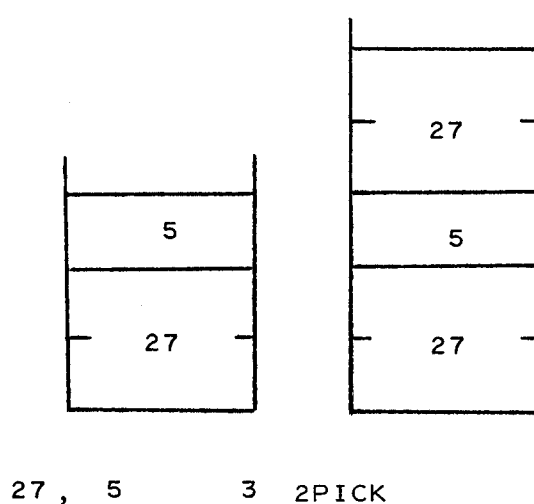
c) 1 DUP 2 OVER + ROT 3DUP 2DROP + * -

- 5) Are the two following sequences identical (i.e. - will the contents of the stack be identical upon completion of each sequence)?

7 4 OVER

7 4 SWAP DUP ROT ROT

- 6) The word `PICK` copies a single-word integer from a location within the stack onto the top of the stack. Define two words `2PICK` and `3PICK` which copy a 2-word entity and a 3-word entity (respectively) from the n^{th} word on the stack onto the top of the stack, where n is the top word on the stack. For example:



6) Continued

	5.8
9	9
3	3
5.8	5.8

5.8 3 9 5 3PICK

9.

All examples and problems up to this point have involved typing in words and numbers on the terminal keyboard and getting the answer printed out immediately. What if we have a sequence of words, such as given in exercise 6 of Chapter 7, to evaluate a given mathematical formula and we want to execute the words many times, each time changing the value of the input variable(s)? Up to this point our only option is to key in the entire sequence of FORTH words comprising the formula each time. Obviously this is unacceptable and it is the purpose of this chapter to introduce the technique whereby a given sequence of words can be "remembered" in the dictionary.

As a simple example say that we want to increment the value of an integer variable (named `J`) by one. The following sequence performs this task:

J @ 1 + J !

Now after keying this in on the terminal a half dozen times we become tired and decide to enter this sequence into the dictionary. First we must assign the sequence a name, since all dictionary entries are identified by the first three characters and count of a user assigned name. If we decide to name it `INCJ` then we may write the colon definition

```

: INCJ J @ 1 + J ! ;
↑   name                               ↑
the colon starts                       the semi-colon terminates
the definition                         the definition

```

First note that the colon starts the definition and the semi-colon terminates the definition. Following the colon is the name that identifies the definition in the dictionary (in this case the first three characters are I N C and the count is 4). Everything following

the name, up to the semi-colon, comprises the definition. When the word `INCJ` is executed *then* the six-word sequence

```
J @ 1 + J !
```

will be executed. Consider the following lines of code as an example:

```
8 J !      [set the current value of J to 8]
INCJ       [increment the value of J by 1]
J @ .      [will print 9]
INCJ       [increment the value of J by 1]
INCJ       [increment the value of J by 1]
J @ .      [will print 11]
```

Perhaps the analogy to keep in mind is the two step compilation/execution sequence of a Fortran program - first you compile the program, subroutine and functions and then you execute the program (which may execute the subroutines or functions, which may execute other subroutines or functions, etc.). In FORTH you may consider keying the colon definition as similar to compilation - the word is entered into the dictionary. After the word has been entered into the dictionary you may then execute the word whenever you wish.

Similar to the concept of one subprogram calling another subprogram, one word in FORTH can execute another word. If we wanted to increment the value of `J` by three then we could define another word named `3INCJ` to perform this:

```
: 3INCJ INCJ INCJ INCJ ;
```

Note however that the above definition will perform the same function as the following definition

```
: 3INCJ J @ 3 + J ! ;
```

that is, increment the value of `J` by 3.

Going back to the Fortran compilation/execution analogy you know that there are two types of errors that may be detected in a Fortran program - compilation errors (missing statement number, invalid syntax, etc.) and execution errors (exponent overflow/underflow, trying to take the square root of a negative number, etc.). Similarly in FORTH, some errors will be detected when the colon definition is *entered* into the dictionary and some errors will not be detected until the newly defined word is *executed*. As an example of the first type of error, if we mistakenly would have keyed in

```
: 3INCJ  XNCJ  INCJ  INCJ  ;
```

↑ error

FORTH will immediately respond with

```
XNCJ  ?Q
```

Since it will not locate the word XNCJ in the dictionary (assuming you have not previously defined a word named XNCJ). This brings up the very important rule:

All words appearing in a colon definition (following the name of the definition and up to the semi-colon) must be either previously defined words or numbers.

This rule is dictated by the fact that the FORTH compiler is a one-pass compiler.

As an example of an execution error assume that we had mistakenly entered

```
: INCJ  1  +  J  !  ;
```

as the original definition of INCJ (we forgot to load the original value of J onto the stack for the addition). FORTH will gladly enter this word into the dictionary however the first time the word INCJ is executed FORTH will respond with

```
INCJ  ?U
```

since the addition operator expects two numbers on the stack.

Returning to the one-pass restriction, it means that the sequence

```
0 VARIABLE J
: INCJ J @ 1 + J ! ;
: 3INCJ INCJ INCJ INCJ ;
```

is valid, however the sequence

```
: INCJ J @ 1 + J ! ;
0 VARIABLE J
: 3INCJ INCJ INCJ INCJ ;
```

is invalid (the definition of J must precede the reference to J in the definition of INCJ). Also, the sequence

```
0 VARIABLE J
: 3INCJ INCJ INCJ INCJ ;
: INCJ J @ 1 + J ! ;
```

is invalid since the reference to INCJ in the definition of 3INCJ precedes the definition of INCJ. Although the one-pass feature of FORTH somewhat restricts the appearance of definitions it greatly speeds up the compilation.

The stack turns out to be the most natural way to pass parameters to a word which is to be executed. Consider the definition of a word we'll call FSQ which forms the square of a floating-point number. If we enter

```
8.0 FSQ F.
```

we expect 64.0 to be printed. Here the number 8.0 is the "parameter" to FSQ and the "result" is left on the stack. The definition of FSQ could be

```
: FSQ 3DUP F* ;
```

and this performs the desired operation. Given this definition of FSQ and the variable declarations

```
7.0 REAL A
8.0 REAL B
1.0 REAL C
```

we can code the expression $(B^2 - 4AC)$ as

```
: DISCR B F@ FSQ 4.0 A F@ F* C F@ F* F- ;
```

where we have chosen the name DISCR to identify this calculation. Entering

```
DISCR F.
```

will print 36.0 as the value of the expression (using the initial values of A, B and C given above).

Similarly, if we enter

```
4.0 A F!
9.0 B F!
2.0 C F!
DISCR F.
```

we should have the value 49.0 printed. We can now use the word DISR that we have defined and write a word to solve the old faithful quadratic equation

$$\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

(Assume at this point that we are dealing only with strictly positive discriminants to avoid worrying about a single root or two imaginary roots.)

```

: 1ROOT B F@ FMINUS DISCR FSQRT F+ A F@ 2.0 F* F/ ;
: 2ROOT B F@ FMINUS DISCR FSQRT F- A F@ 2.0 F* F/ ;
: QUAD 1ROOT F. 2ROOT F. ;

```

Executing QUAD (with A = 4.0, B = 9.0, C = 2.0 from above) should print

```

-0.25      -2.0

```

as the roots of the equation. The new words introduced in the above are FMINUS (which negates a floating-point number) and FSQRT (which evaluates the square root of a floating-point number).

A useful word that deserves mentioning here is the SET word which can often times be used instead of a colon definition to set a flag. Consider the definition

```

0 VARIABLE ?PLOT

```

which we use as a logical flag (Section 7.5) to indicate whether or not we want to plot some points. A simple way to set or reset this flag would be

```

: PLOTON 1 ?PLOT ! ;
: PLOTOFF 0 ?PLOT ! ;

```

However a faster and more efficient method of setting the integer ?PLOT to a specific value is to enter the definitions

```

1 ?PLOT SET PLOTON
0 ?PLOT SET PLOTOFF

```

Executing the word PLOTON will set the value of ?PLOT to 1, regardless which definition is used.

The general form for the SET definitions is

```

<integer-value> <address> SET <name>

```

and then executing the word <name> will store the <integer-value>

at the specified <address>. (Recall from Section 7.1 that executing the word ?PLOT will push onto the stack the address of the variable). If we had a definition

```
0 CONSTANT FLAG
```

then we could define the words

```
0 ' FLAG SET FLAGON
1 ' FLAG SET FLAGOFF
```

to turn the flag on and off.

If one wanted to turn on two flags at the same time then the word 2SET may be used. Its form is

```
<value1> <address1> <value2> <address2> 2SET <name>
```

and then executing the word <name> will store <value1> at <address1> and <value2> at <address2>. For example, using the previous definitions of ?PLOT and FLAG we could write the definitions

```
: BON 1 ?PLOT ! 1 ' FLAG ! ;
: BOFF 0 ?PLOT ! 0 ' FLAG ! ;
```

or equivalently,

```
1 ?PLOT 1 ' FLAG ! 2SET BON
0 ?PLOT 0 ' FLAG ! 2SET BOFF
```

EXERCISES - CHAPTER 9

- 1) Define a word named I**4 that will raise the single-word integer on top of the stack to its 4th power. There are two obvious ways to code this word. Verify that $2^4 = 16$, $3^4 = 81$, $4^4 = 256$, $5^4 = 625$, etc. Which method is preferable?

10. PROGRAM CONTROL

In the discussion of the colon definition in the previous chapter the words comprising the definition were executed sequentially. For example, in the definition

```
      ; QUAD  1ROOT  F.   2ROOT  F.   ;
```

first the word 1ROOT is executed, then the word F. is executed, then the word 2ROOT is executed and finally the word F. is executed.

Often times we want to be able to control the flow of execution through a colon definition based on certain programmatic decisions. In Fortran you may use DO loops and IF statements to control the flow of execution through a program and FORTH contains similar control structures, described in this chapter.

10.1 DO LOOPS

The FORTH DO loop may be used within a colon definition to repeatedly execute a sequence of words. There are two forms provided, depending whether the loop index is to increment by +1 each time through or whether the loop index is to change by a programmer specified value:

```
<limit-value> <starting-value> DO    <words>    LOOP
```

```
<limit-value> <starting-value> DO    <words>    <increment-value> +LOOP
```

<starting-value> is a single-word integer value specifying the value of the loop index the first time through the loop.

<limit-value> is a single-word integer value specifying the upper limit of the loop index. If the loop index is *increasing*, the loop will terminate when the loop

index *reaches* this <limit-value>. When the loop index is *decreasing*, the loop will terminate when the loop index *passes* this <limit-value>.

<words> are the words to be executed each time through the loop.

<increment-value> is a single-word integer value specifying the value by which the loop index is to be incremented *or* decremented each time through the loop. If this <increment-value> is specified then the word +LOOP must terminate the DO loop.

As in Fortran, a DO loop in FORTH will always be executed at least once. Some examples should help clarify the above descriptions.

LOOP SPECIFICATION						LOOP INDEX VALUES	
5	1	DO	<words>	LOOP		1, 2, 3, 4	} identical loops
5	1	DO	<words>	1	+LOOP	1, 2, 3, 4	
1	5	DO	<words>	-1	+LOOP	5, 4, 3, 2, 1	
-8	-6	DO	<words>	-1	+LOOP	-6, -7, -8	
11	3	DO	<words>	2	+LOOP	3, 5, 7, 9	
-3	-11	DO	<words>	+2	+LOOP	-11, -9, -7, -5	
50	25	DO	<words>	5	+LOOP	25, 30, 35, 40, 45	
0	1000	DO	<words>		LOOP	1000	
-99	-99	DO	<words>	-1	+LOOP	-99	

Before continuing on make certain that you understand these examples and proceed to the exercises at the end of the chapter and work the first exercise.

In order to access the loop index while executing a DO loop the word I must be executed. Executing this word pushes onto the stack the current single-word integer value of the loop index. In this respect the word I acts like a CONSTANT and not a VARIABLE. For example we can define a word named PRNT

```
: PRNT 5 1 DO I . LOOP ;
```

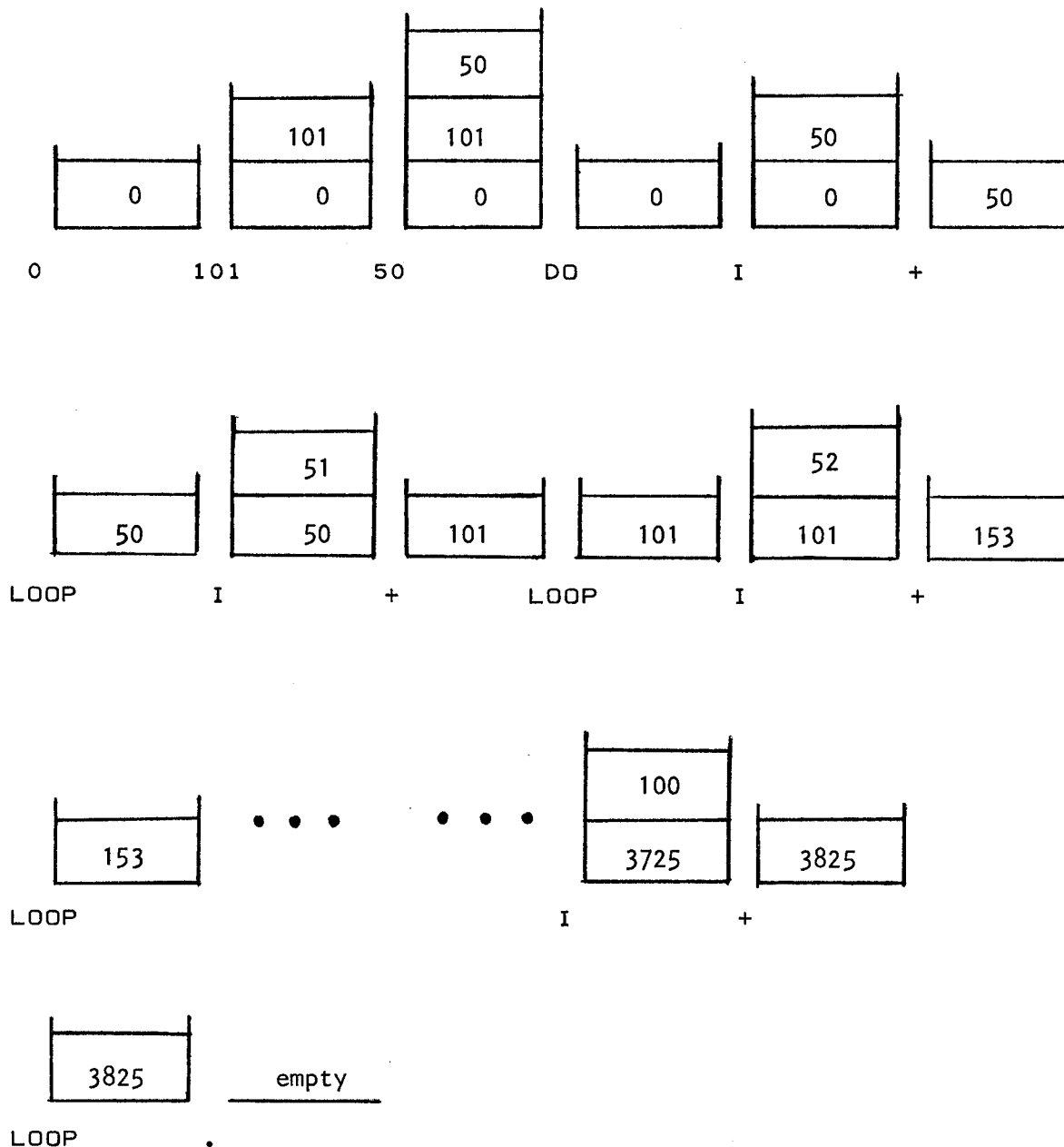
and executing the word PRNT will print 1 2 3 4 on the terminal. If we wanted to form the sum of the integers from 50 through 100 (inclusive) and then print the result we could define a word named SUM

```
: SUM 0 101 50 DO I + LOOP . ;
```

and executing the word SUM will print 3825. Note that in this example we initially push 0 onto the stack to initialize the sum. We then add each value of the loop index to the top number on the stack, leaving the result on top of the stack for the next time through the loop. When the loop terminates, the top number on the stack is the sum and we may then print this value. This is a good example of the usefulness of keeping a temporary result on the stack instead of storing and loading the value in a temporary variable. Convince yourself that the following word performs the identical function as SUM

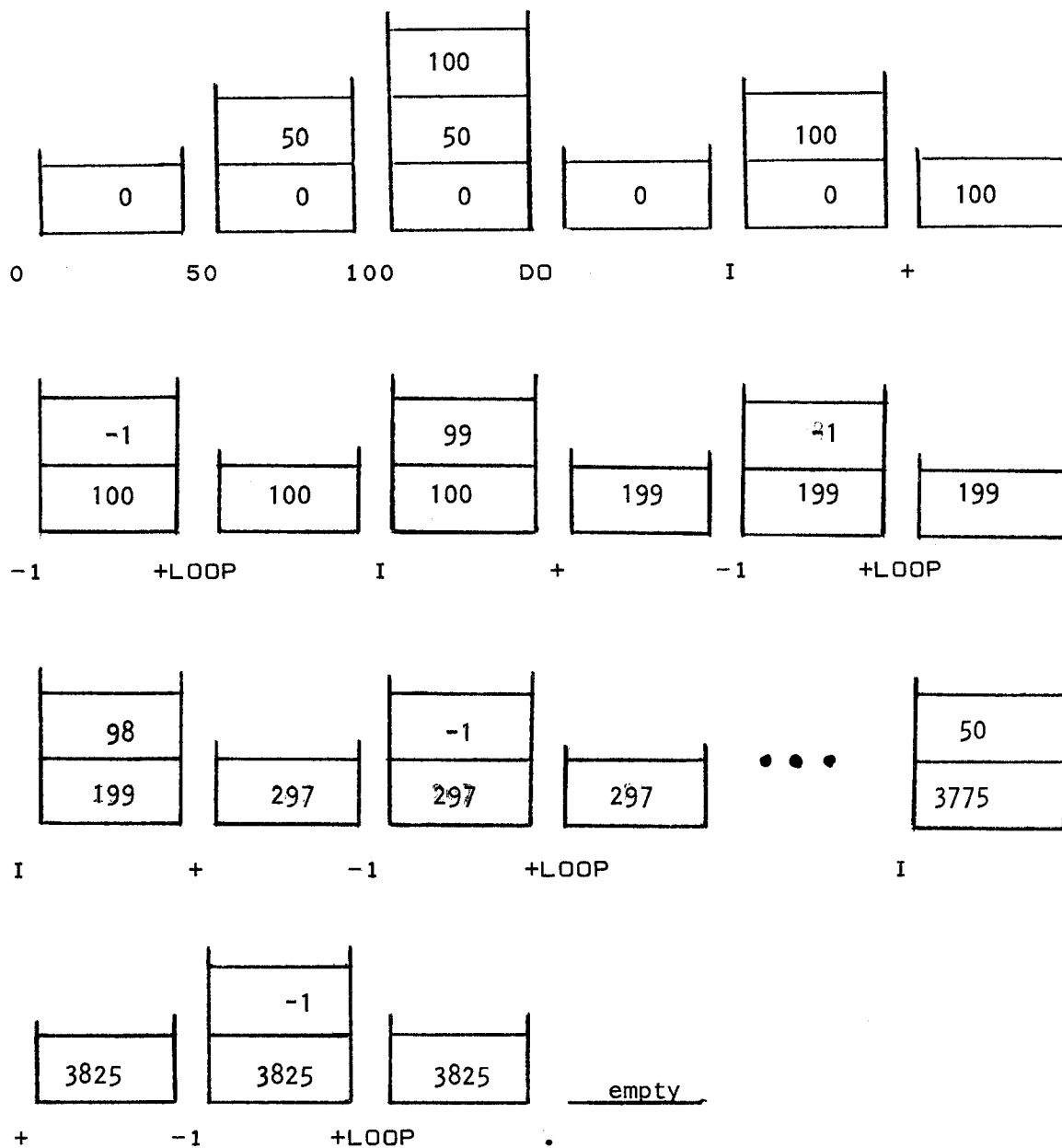
```
: SUM1 0 50 100 DO I + -1 +LOOP . ;
```

The stack operations involved when the word SUM is executed are as follows:



Note that the <starting-value> and <limit-value> are popped off the stack by the word DO and are then stored internally within FORTH. Note also that the word LOOP does not manipulate the stack (and is shown in the above simply for clarity).

To show the stack operations involved when using the word `+LOOP` to terminate a `DO` loop consider the word `SUM1` defined above



The word `+LOOP` pops the <increment-value> off the stack each time it is executed.

DO loops in FORTH may be nested and in order to access the loop index from within a nested loop the words I, J and K are used as follows:

- I - accesses the loop index from the innermost loop,
- J - accesses the loop index from the loop outside of the innermost loop,
- K - accesses the loop index from the loop two levels outside of the innermost loop.

The words I J K act as CONSTANTS in that one simply executes the word in order to push the value of the appropriate loop index onto the stack. However, it is *not* possible to store new values in these words by executing, for example, ' I ! . In other words, the current value of the loop index may be read only.

DO loops in FORTH are not limited to a maximum nesting of three, as might be indicated from the discussion above, however the gory details of further nesting is beyond the scope of this primer.

If the word LEAVE is executed within the range of a DO loop this will force the loop to terminate when the next LOOP or +LOOP is executed. This provides a handy method of terminating a loop before the specified <limit-value> is reached.

The current value of a loop index is accessible *only* within the colon definition which contains the words DO and LOOP. For example

```
      : ISQ I DUP * ;  
      : ILOOP 10 0 DO ISQ LOOP ;
```

is not allowed since I (the current value of the loop index) is available only within the word ILOOP and not within words called by ILOOP (such as the word ISQ).


10.2 BEGIN-END LOOPS

This control structure may be used within a colon definition to repetitively execute a sequence of words until a specified logical condition is true. The format of the loop is

```
BEGIN  <words>  <logical-condition>  END
```

The most frequent use of this structure is to "wait" for a certain condition to occur. For example, basic FORTH provides a word named `?TER` and when executed it pushes a value of true onto the stack (a non-zero integer value) only if the operator has entered a character on the terminal, otherwise a value of false (zero) is pushed onto the stack. If, at some point of a program, you wish to wait for the operator to enter a character, you could write

```
  : <name>  ...  BEGIN  ?TER  END  ...  ;
```


this loop is executed continually
until a character is entered on
the terminal.

As another example assume that we want to terminate a loop only when the value of a floating-point variable named `DELTA` is less than 0.01:

```
FLOATING
```

```
0.0  REAL  DELTA
```

```
  : <name>  ...  BEGIN  <words>  DELTA F@ 0.01  F<  END  ...  ;
```

This example is typical of many numerical iterations where one is waiting for a value to converge to some limiting value.

As another example let us code the word `SUM` from the previous section without using the `DO` loop:

```

0  CONSTANT  INDEX
:  INCINDEX  INDEX  1  +  ' INDEX !  ;
:  SUM2  50  ' INDEX !  0  BEGIN  INDEX  +  INCINDEX
      INDEX  100  >  END  .  ;

```

Executing the word `SUM2` results in 3825 being printed, identical with the execution of the word `SUM`, however note how much more work is involved by not using the `DO` loop. Note that the <logical-condition> is popped from the stack by `END`.

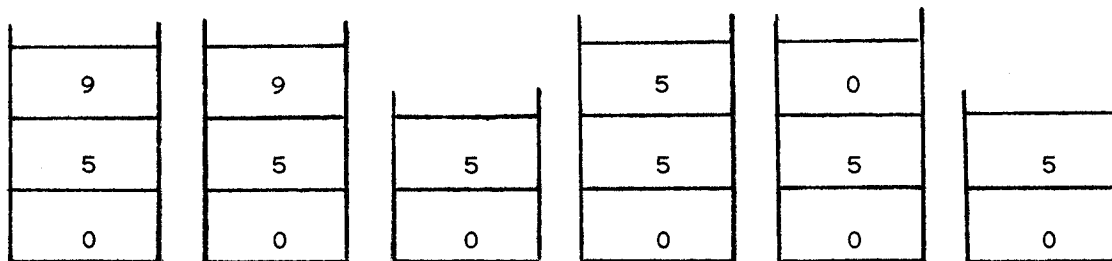
Finally, consider a word named `STKPRNT` which proceeds down the stack, printing each number, until a zero is encountered.

```

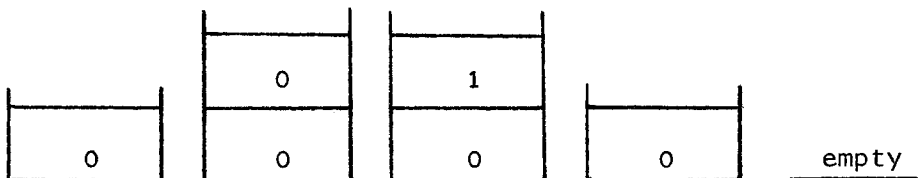
:  STKPRNT  BEGIN  .  DUP  0=  END  DROP  ;

```

Executing `0 5 9 STKPRNT` would result in `9 5` being printed and the following stack operations:



0 5 9 STKPRNT BEGIN . DUP 0= END



. DUP 0= END DROP

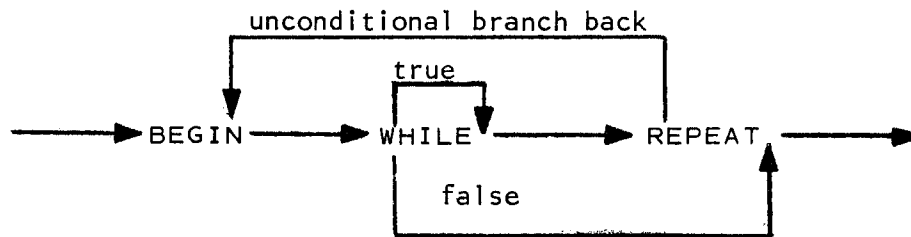
(The word `BEGIN` does not manipulate the stack and is shown above simply for clarity).

10.3 BEGIN-WHILE-REPEAT LOOPS

Often it is desirable to terminate a loop at some point *within* the loop and not at the end (as BEGIN-END does). The control structure provided for this is

```
BEGIN <words-1> <logical-condition> WHILE
                                <words-2> REPEAT
```

First <words-1> are executed and the <logical-condition> is then evaluated. If true (non-zero) then <words-2> are executed and REPEAT will transfer control back to BEGIN and <words-1> are evaluated. If false (zero) then WHILE transfers control to the words following REPEAT. As long as the <logical-condition> is true, <words-1> and <words-2> are executed. However, as soon as the <logical-condition> is false, the loop is executed immediately (<words-2> are not executed after WHILE encounters a false). Graphically, this structure appears as



After reading the next section, one should understand that the BEGIN-WHILE-REPEAT structure could be implemented as

```
BEGIN
    <words-1> <logical-condition>
    IF <words-2> 1
    ELSE 0 THEN
END
```

where the 1 and 0 left on the stack by the IF-THEN-ELSE are simply flags for END to either terminate or continue looping.

10.4 IF - THEN - ELSE STATEMENT SELECTION

This control structure allows the program flow to branch in one or two directions depending on the value of a logical condition. This version of the IF statement is more powerful than either the arithmetic IF or the logical IF in Fortran since an ELSE clause is provided.

The format of the IF statement is either

```
<logical-condition> IF <true-part> THEN
```

```
<logical-condition> IF <true-part> ELSE <false-part> THEN
```

where the words comprising the <true-part> will be executed only if the <logical-condition> is true (non-zero), otherwise the words comprising the <false-part> will be executed (if the ELSE <false-part> clause is specified).

As an example of a word using the ELSE clause consider a word named SIGN which prints "POS" or "NEG" on the terminal depending whether the integer on top of the stack is positive or negative:

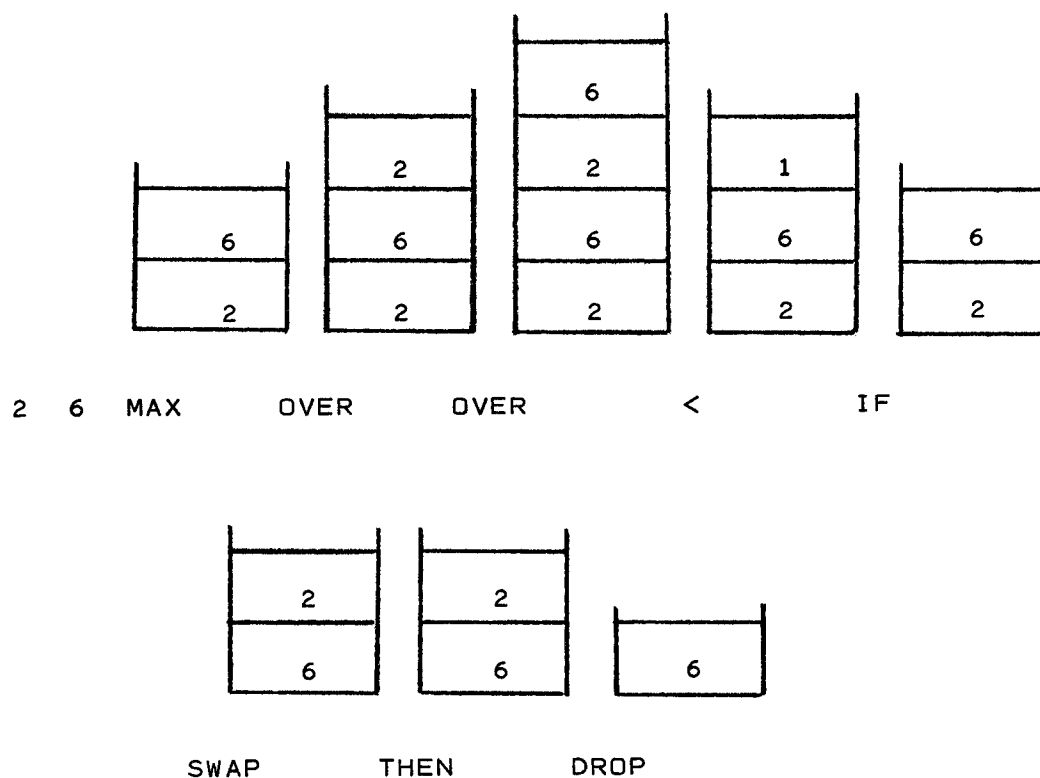
```
: SIGN 0 < IF ." NEG" ELSE ." POS" THEN ;
```

(character output is described in Section 14.1).

As another example consider the following definition of the word MAX (which was described in Chapter 8):

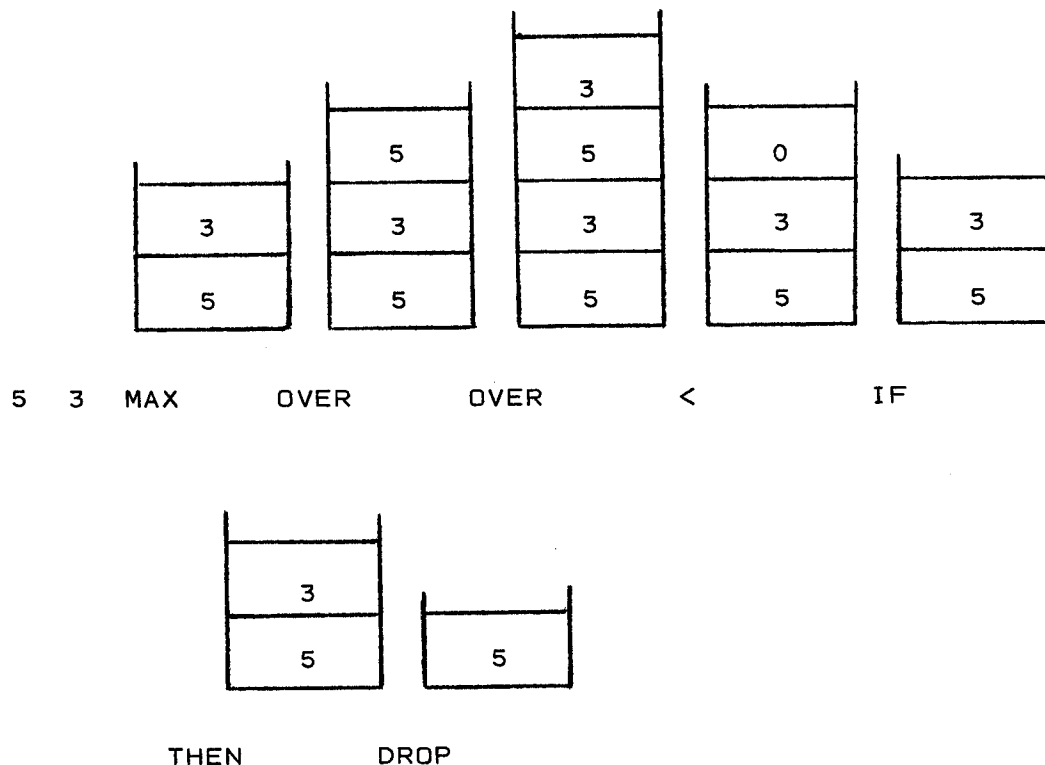
```
: MAX OVER OVER < IF SWAP THEN DROP ;
```

The stack operations involved in executing 2 6 MAX are:



The word `>` generates the <logical-condition> of 1 (true) and therefore the <true-part> is executed (the word `IF` pops the 1 off the stack). The word `THEN` does not manipulate the stack and is shown above simply for clarity.

If we were to execute 5 3 MAX the stack operations would be:



Since the <logical-condition> is 0 (false) and since an ELSE clause is not specified the word THEN is executed immediately after the word IF.

It is important to note that the IF statement pops the <logical-condition> off the stack. If you want to preserve the <logical-condition> for later use then you must DUP it prior to executing the IF. Another important point to note is that the stack must be the same (i.e. - contain the same number of words) after executing either the <true-part> or the <false-part>. Exercise 6 at the end of this chapter illustrates this point.

EXERCISES - CHAPTER 10

1) What values will the loop index take on in each of the following DO loops?

- a) 2900 2898 DO <words> LOOP
- b) -5 -4 DO <words> -1 +LOOP
- c) -6 -2 DO <words> -2 +LOOP
- d) -2 -3 DO <words> LOOP
- e) -2 -3 DO <words> -1 +LOOP
- f) 6 18 DO <words> -6 +LOOP
- g) 18 6 DO <words> 6 +LOOP
- h) 0 -1 DO <words> LOOP

2) Write a word to form the product of the even numbers from 2 to 10 (inclusive) and print the result. Use a DO loop with a positive increment.

3) Redo exercise 2, this time using a DO loop with a negative increment.

4) Recode the words INCINDEX and SUM2 in Section 10.2 using the definition

O VARIABLE INDEX

5) Use the colon definition to define the words:

D MAX D MIN D = D < D > D O = D O <

- 6) Extend the word `SIGN` given in Section 10.3 to print `"P"`, `"N"` or `"Z"` depending whether the integer on top of the stack is positive, negative or zero.
- 7) Define a word named `EX` which prints the top `n` words of the stack, non-destructively -- that is, the contents of the stack must not be destroyed (the printing word `.` destroys the number that it prints). For example

```
88 23 2 EX - .
```

should print 23 88 65

- 8) Define a word named `VINIT` that will initialize each element in the vector

```
19 ( )DIM VEC
```

to its subscript value. That is, element 0 should be set to 0, element 1 should be set to 1, ..., element 19 should be set to 19.

- 9) Define three words named `()`, `2()`, `3()`. that will print out the first `n` values of a single-word integer vector, a double-word integer vector and a floating-point vector, respectively (`n` is the top word on the stack, the starting address of the vector is the second word on the stack). Test the word `()` on the vector in the previous problem.
- 10) Define a word named `3VBUBSORT` that will sort a vector of floating-point numbers into ascending order using the bubble sort algorithm. This algorithm is one of the simplest (and least efficient) techniques for sorting a string of numbers into order and is defined as follows:

To sort the N values $X(0), X(1), \dots, X(N-1)$

```
For L = 1, 2, ..., N-1
  For M = N-1, N-2, ..., L
    If  $X(M-1) > X(M)$ 
      Then Swap  $X(M)$  and  $X(M-1)$ 
```

Note that this algorithm uses two (nested) DO loops, one with an increasing loop index and one with a decreasing loop index.

Assume that the top word on the stack is N . Use the definitions

```
5  3()DIM  VC
29.7  0  VC  F!
-8.2  1  VC  F!
-1.9  2  VC  F!
4.5   3  VC  F!
0.52  4  VC  F!
-8.3  5  VC  F!
```

and then execute

```
6  3VBUBSORT
```

Use the word `3()`. from the previous problem to print the sorted vector.

11. BLOCK I/O

Using the methods introduced up to this point, to enter a word into the dictionary we type in the definition through the terminal and the word is immediately entered into the dictionary. This method has two obvious disadvantages: (a) we are restricted to one line of terminal input (72 characters) per definition; (b) if we do not have a hard copy printout of our terminal input/output and we forget a previously entered definition we are out of luck as far as recalling the listing of the definition. Both of these restrictions are overcome by using "blocks" as described in this chapter.

FORTH divides the secondary storage (disc or tape) into separate distinguishable units called blocks. In KPN0 Varian FORTH systems (with a 5 Megabyte disc) each block is identified by its block number, an integer between 0 and 4895 (inclusive). Each block contains 1024 characters. (Blocks may contain data other than characters however in this primer we will be interested in blocks of characters only).

Certain blocks are permanently allocated, as shown below:

<u>Block #</u>	<u>Usage</u>
0 - 7	Bootstrap & binary loaders
8 - 199	FORTH
200	Resident Loader Block, to be set by the user.
201 - 4895	Available for the user.

Additionally, executing

130 LOAD

will list on the terminal the block allocation within blocks 8-199.

From the block allocation scheme listed above, the user is allowed to do anything he wants with blocks 200-4895 (but must *not* modify any of blocks 0-199).

Even though a block is nothing more than 1024 consecutive characters, certain routines (the text editor and the block lister) divide these 1024 characters into 16 lines of 64 characters/line. The purpose of this arbitrary division is simply to facilitate the reading and printing of the characters in a block. This division of a block into lines in no way affects the format of the data on magnetic tape or disc. The 16 lines in a block are referenced as line 1, line 2, ..., line 16. One common error made by newcomers to FORTH is to assume that there is a blank between the last column of a given line and the first column of the next line. This is false as the block consists of 1024 *consecutive* characters - no special characters are inserted before or after each line, as the breakdown of a block into lines is for reading and printing only.

12. TEXT EDITOR

Knowing that a block of secondary storage can contain 1024 characters of FORTH code we now need a way to efficiently manipulate a block of characters. This manipulation is the function of the text editor. Facility in using the editor is a must in order to proceed onward in this primer as all examples and exercises from this point on will require the use of blocks to contain your program code.

12.1 SPECIAL CHARACTERS AND TERMINOLOGY

The following special characters will be referred to in the descriptions that follow:

<i>space</i>	press the space bar on keyboard.
<i>carriage-return</i>	press the "RETURN" key on keyboard.
<i>line-feed</i>	press the "LINE-FEED" key on keyboard.
<i>rubout</i>	press the "RUBOUT" key or the "DELETE" key on keyboard.
<i>tab</i>	press the "CNTL" key and the "I" key simultaneously.
<i>break</i>	press the "BREAK" key on keyboard.
<i>escape</i>	press the "ESC" key or the "ALT MODE" key on the keyboard.
<i>control-N</i>	press the "CNTL" key and the "N" key simultaneously.

The following symbols will be used to denote frequently used entities:

<i><block#></i>	refers to a block number, which is a single-word integer in the range 0 through 4895 (inclusive).
<i><line#></i>	refers to a line number, which is a single-word integer in the range 0 through 16 (inclusive).
<i>text buffer</i>	is a 64 character buffer used to hold a single line. Certain editor commands will move a line into the text buffer and other editor commands may move the contents of the text buffer into a certain line of the block.

12.2 COMMAND DESCRIPTIONS

Describing the editor is a somewhat boring task since all one can do is verbally define each command - examples are generally impossible to provide since the editor is such an interactive, terminal-oriented program. The only way to appreciate this interactiveness and to understand each of the commands below is to try executing each command after reading its description. This chapter may also be used later as a reference when using the editor.

<block#> EDIT will load the editor program into the dictionary, if it is not already in the dictionary. The specified block is then read into a core buffer from secondary storage (tape or disc) and is listed on the terminal. You may stop this listing by pressing any terminal key. NOTE: When using the file system (Section 13.2) the *first* invocation of the editor must not be preceded by a <block#>. This is because after the editor is loaded but before any block number may be specified, one must first execute a FILE command to designate which file is to be edited. A typical invocation of the editor with the file system would be

```
EDIT                (load the editor)
FILE <file-name>    (specify which file)
<block#> EDIT        (edit a block within the
                      file)
```

A will edit the Alternate block. FORTH has two block buffers in memory and if one wants to transfer some lines between two blocks, execute

```
<block#1> EDIT      (read block#1 into memory)
<block#2> EDIT      (read block#2 into memory)
... HL              (hold lines from block#2)
A                   (switch to block#1)
... IL              (insert lines into block#1)
A                   (switch to block#2)
```

B

will Begin entering lines from the keyboard into the block being edited, starting at the line specified by <line#>. FORTH will print each line number and then wait for you to key in the contents of the line, terminating each line with a *carriage-return*. This process continues until either line 16 (the last line in the block) is entered or until you key in a *break* as the first character of a line. While keying in each line the following special characters may be used:

rubout - deletes the preceding character (backspaces).

break -erases the entire line allowing you to start the line over again (this assumes that the *break* was keyed in other than as the first character of a line - when a *break* is keyed in as the first character the Begin mode is terminated, as mentioned above).

tab - enters 5 spaces into the line.

Note that each line entered by using the Begin command replaces the previous contents of that line.

Example: 2 B will start replacing at line 2 of the block. If 4 lines are entered by the operator then lines 2, 3, 4, & 5 are replaced. Lines 1 and 6-16 will not be modified.

<line#> BI

will Begin Inserting lines from the keyboard into the block being edited, starting after the line specified by <line#>. This command is similar to the B command with the following exception: entering 3 lines with the command '8 B' will replace lines 8, 9 & 10 with the new lines entered by the operator - lines 1-7 and lines 11-16 are not modified and remain intact; entering 3

lines with the command '8 BI' will replace lines 9, 10 & 11 with the new lines entered by the operator - lines 1-8 are not modified, however, the contents of old line 9 are now in line 12, the contents of old line 10 are now in line 13, the contents of old line 11 are now in line 14, ..., the contents of old line 13 are now in line 16 and lines 14, 15 & 16 have been discarded. Thus the B command *replaces* lines in a block, overwriting the previous contents of each line; the BI command *inserts* new lines into a block. Since each block contains exactly 16 lines, when you insert a new line one of the old lines has to be removed - the algorithm used by BI is to push each succeeding line down one line, effectively pushing line 16 out of the block.

<source-block#> <destination-block#> B-MOVE

Moves the block specified by <source-block#> This action is identical to the MOVEBLOCKS command (described in the next section) with <#-of-blocks>

Example: 500 509 B-MOVE }
 500 509 L MOVEBLOCKS }
 both commands move block 500
 to block 509.

<block#> CHANGE

Changes the block number of the block currently being edited to <block#>.

Example: 292 EDIT }
 208 CHANGE } effectively moves block
 292 to block 208.

CLEAR

will initialize the block being edited to blanks (i.e. - the entire block is erased). This command is used to clear a block prior to entering new code into the block. A block of all blanks is considered in use and will be printed on a DOFORTH listing.

<line#> D DDeletes the line specified by <line#>. The original contents of the line are placed in the text buffer and may therefore be moved to another line with the I or R commands. All lines following <line#> are moved up one line and line 16 is filled with blanks.

Examples: 9 D will delete line 9, moving
 lines 10-16 into lines 9-15
 and line 16 then is filled
 with spaces.

9 D } will delete line 9 and then
4 I } insert the original contents
 of line 9 between lines 4 & 5.
The net effect of these two
commands is that lines 1-4
are unmodified, line 5 contains
the original contents of line
9, lines 6-9 contain the
original lines 5-8 and lines
10-16 are unmodified.

<line#1> <line#2> DL DDeletes the Lines <line#1> through <line#2>
(inclusive) of the current block being edited.
This command is identical to a string of D
commands. At the completion of this command the
first line in the sequence will be in the text buffer.

Example: 7 D 6 D 5 D }
 5 7 DL

these two command strings perform the
same function, namely deleting lines
5 through 7 of the current block.

<line#> E Erases the line specified by <line#>, filling the
line with blanks. The original contents of the
line are first placed in the text buffer and may
therefore be moved to another line with the I
or R commands.

Example: 5 E } erases line 5 and then inserts
 9 I } the original contents of line
 5 between lines 9 & 10, moving
 lines 10-15 down into lines
 11-16. The original contents
 of line 16 are lost.

ERASE-CORE

Prevents the block currently being edited from being written onto secondary storage (disc or tape) by marking both of FORTHs block buffers empty. This allows you to change your mind after editing a block, provided you have not specifically FLUSHed the block onto secondary storage or edited other blocks since making the changes.

Example: 250 EDIT } edits block 250 with the
 (changes to } specified changes and
 block 250) } then effectively ignores
 ERASE-CORE } these changes. After
 executing the ERASE-
 CORE the contents of
 block 250 will be the
 same as prior to executing
 the 250 EDIT.

FLUSH

Forces the writing onto secondary storage (disc or tape) of both of FORTHs block buffers. Once a block is written onto secondary storage the previous contents of the block are lost. Normally FORTH does not write a block buffer onto secondary storage until the buffer space is needed for another purpose, however this command is a way of guaranteeing that an edited block replaces the previous contents of the block (in case the system were to fail between the editing of the block and FORTHs normal buffer output).

<line#> H

Holds the line specified by <line#> in the text buffer. A copy of this line may then be moved to another line with the I or R commands. Unlike the D command, the H command does not delete the line when it is placed in the text buffer.

Examples: 2 H } will hold line 2 in the text buffer.

2 H } will hold line 2 in the text
8 R } buffer and then replace line 8
with this copy of line 2.

Effectively we have copied
line 2 into line 8.

<line#1> <line#2> HL

Holds Lines <line#1> through <line#2> (inclusive) of the current block being edited. These lines may then be moved to another block with the IL command. This hold is not a true hold as only the line numbers are remembered and no text buffers are used. After executing an HL command the user must *not* issue the B, BI or M commands prior to executing the IL command. Also, the HL, IL commands may *not* be used to move lines within a single block.

Example: 258 EDIT } will move lines 5, 6 & 7
5 7 HL } of block 258 into lines
261 EDIT } 10, 11 & 12 of block 261.
12 D } Note that the previous
11 D } contents of lines 10, 11
10 D } & 12 are deleted prior to
9 IL } entering the new lines.
Also note that lines 5, 6
& 7 of block 258 have not
been deleted from block 258.

<line#> I

Inserts the contents of the text buffer after the line specified by <line#>. The lines following <line#> are each moved down one line and hence line 16 is lost. <line#> may be 0 in which case the new line becomes line 1.

Examples: 12 I } will insert the contents of the text buffer after line 12. The new line becomes line 13 and the original lines 13-15 become lines 14-16. Line 16 is lost.

12 D }
12 I } effectively swaps lines 12 and 13. This is so because the delete places the original line 12 in the text buffer and moves lines 13-16 into lines 12-15. The insert then places the original line 12 from the text buffer into line 13, moving lines 13-15 into lines 14-16. Hence the original line 13 is now in line 12 (from the delete) and the original line 12 is now in line 13 (from the insert).

<line#> IL

Inserts the Lines that were held by the most recent HL command into the current block being edited, effectively moving a copy of the lines from the prior block into the current block. The lines in the current block, following <line#>, are moved down as required to make room for the new lines.

As usual, extra lines will be lost when moved down from line 16. After executing an HL command the user must *not* issue the B, BI or M commands prior to executing the IL command. Also, the HL, IL commands may *not* be used to move lines within a single block.

L List the block being edited. On a CRT terminal the screen will be erased prior to listing the block. You may stop the listing by pressing any terminal key.

<line#> M Modifies the line specified by <line#>. This command is probably the most frequently used editor command therefore acquaintance with this command is a necessity. Since this command only modifies the specified line no other lines in the block are affected. The following buffers are used by the editor to execute this command: (note: this command is in no way as complicated to use as it is to describe).

Reference Line - A copy of the original contents of the specified line. It is listed when this command is executed.

Control Line - This line is used to enter the special input character codes into. On a CRT terminal this line is directly beneath the Reference Line, while on a printing terminal (such as a teletype) this line is coincident with the Output Line.

Output Line - A copy of the final contents of the modified line being constructed. This line is built up as you proceed, character by

character, through the Reference Line and after normal termination of the M command this line replaces the original contents of the specified line.

The M command functions by proceeding character by character through the line, adding, deleting or replacing characters, building a new copy of the line. The following characters have special meanings when executing the M command. Any character not defined below, when keyed in, will be transferred into the output line.

space - Transfers the next character from the Reference Line into the Output Line (i.e. - copies one character from the old version of the line into the new version being constructed) and then moves to the next character position in the Reference Line.

tab - Transfers the next 5 characters from the Reference Line into the Output Line (equivalent to typing in 5 *spaces*).

carriage-return - Transfers the remainder of the Reference Line into the Output Line. This terminates the M command and the contents of the Output Line are then moved into the specified line of the block.

rubout - Skips over the next character in the Reference Line, effectively "deleting" the character from the new version of the line.

Control-N - Transfers one blank into the Output Line without advancing in the Reference Line, effectively "adding" a blank into the new version of the line.

- \overline{L} - (a *Control-L* followed by a single character).
Transfers characters from the Reference Line into the Output Line up to but not including the character c. This action is a handy way to leave in all characters up to specific character without having to continually enter *spaces* or *tabs*.
- \overline{Kc} - (a *Control-K* followed by a single character).
Skips through the Reference Line until the character c is reached. No characters are transferred into the Output Line. This action effectively kills all characters, up to the character c, from the new version of the line.
- Control-S* - Skips to the next word, transferring the characters that are skipped into the output line.
- Control-D* - Deletes to the next word. No characters are transferred into the Output Line.
- break* - Aborts the current M command leaving the original contents of the specified line intact. The "?S" abort message is printed (Appendix B).
- line-feed* - Starts transferring characters from the keyboard into the Output Line, effectively "adding" characters into the new line being formed. Every character keyed in is moved to the output with the exception of the following special characters:
- rubout* - Deletes the previous character in the Output Line, effectively "backspacing" one character.
- break* - Deletes the entire accumulated output line. Additionally, if a

break is entered as the first character of the output line then the M command is aborted, leaving the original contents of the specified line intact.

tab - Transfers 5 spaces into the Output Line (equivalent to typing in 5 spaces).

carriage-return - Terminates the M command and the contents of the Output Line are transferred into the specified line of the block. Note that all remaining characters in the Reference Line, following where the *line-feed* was keyed in, are effectively deleted.

escape - Terminates the "add character" mode that was initiated by the *line-feed*, returning "control" to the Reference Line.

Whenever characters are added or deleted, within a line, by using the M command, the characters that follow in the line will be shifted left or right. If characters are added to a line then the remaining characters are shifted right and any characters beyond the 64th character position are lost (they do *not* get moved into the next line). If characters are deleted from a line then the remaining characters are shifted left and spaces are added through the 64th character position. This action is similar to what happens with line 16 when entire lines are added or deleted from a block. On CRT terminals (as opposed to printing terminals) a cursor is displayed to help you visualize where you are in either the Reference Line or the Output Line as you move through a line.

N

<line#1> <line#2> O

will edit the Next block.

will perform a character by character logical-OR of the two lines, printing the result and leaving it in the text buffer. The logical-OR of any character with a blank is the character itself.

P

<line#> R

will edit the Previous block.

Replaces the line specified by <line#> with the contents of the line buffer.

Example: 2 T } will type line 2 and then
 14 R } move the copy of line 2 into
 line 14. Lines 2, 15 and 16
 are not modified.

<line#> T

Types the line specified by <line#> and places the line in the text buffer.

Example: 2 T } will type line 2 and then move
 14 I } the copy of line 2 into line
 15. The original contents of
 line 15 are moved to line 16
 and the original contents of
 line 16 are lost.

ZERO

will initialize the block being edited to zeros (i.e. - the entire block is erased). A block of all zeroes is considered an unused block and will not be printed on a DOFORTH listing. This command is often used to clear a block that was in use but is no longer needed.

12.3 BLOCK EDITOR

The text editor described in the preceding section operates on *lines* of text and manipulates these lines within a given block or between two different blocks. This section describes an additional editor, referred to as the Block Editor, which manipulates entire blocks or groups of blocks, regardless what is contained within the block, be it characters or data. Our usage of the block editor will assume that the blocks contain character data, however, the block editor commands will also operate on blocks of data.

In order to load the block editor into the dictionary the text editor must first be loaded into the dictionary. The text editor is loaded by executing the sequence (refer to Section 12.2)

```
<block#> EDIT
```

Following this, one loads the block editor by executing the sequence

```
199 LOAD
```

The following commands are then available to manipulate groups of blocks:

```
<starting-block#> <ending-block#> CLEARBLOCKS
```

will fill each block from <starting-block#> through <ending-block#> (inclusive) with blanks. This command is similar to the text editor CLEAR command.

```
Example: 281 294 CLEARBLOCKS
```

will fill block 281 through 294 with blanks.

```
<block#1> <block#2> <#-of-blocks> EXCHANGE
```

will exchange (swap) the number of blocks specified by <#-of-blocks> between the blocks specified by <block#1> and <block#2>.

```
Example: 500 600 3 EXCHANGE
```

will exchange blocks 500 and 600, blocks 501 and 601 and blocks 502 and 602.

```
<source-block#> <destination-block#> INSERT
```

will insert the block specified by <source-block#> immediately following the block specified by <destination-block#>. The blocks starting with <destination-block#> are all moved down one block, until the first empty block is encountered. (A block is considered empty if it contains all zeros). If there are no empty blocks within 50 blocks of <destination-block#> then no block movement

takes place and a message is printed. If an empty block is located then the block number of this empty block is printed.

Example: 325 408 INSERT

will move the contents of block 325 into block 408. Assuming that block 412 is the first empty block following block 408, then the following block movement will take place:

411 --> 412

410 --> 411

409 --> 410

408 --> 409

325 --> 408

Additionally, the message

BLK USED: 412

will be output.

<source-block#> <destination-block#> <#-of-blocks> MOVEBLOCKS

will move the number of blocks specified by <#-of-blocks> starting from the block specified by <source-block#> into the blocks starting at the block specified by <destination-block#>. The original contents of the destination blocks are overwritten by the new contents. The blocks are moved from first to last, therefore no overlap is allowed. This means that if <destination-block#> minus <source-block#> is less than <#-of-blocks>, information will be destroyed.

Example: 380 385 3 MOVEBLOCKS

will move block 380 into block 385, block 381 into block 386, and block 382 into block 387. The original contents of blocks 380, 381 and 382 are not modified.

<starting-block#> <ending-block#> ZEROBLOCKS
will fill each block from <starting-block#> through <ending-
block#> (inclusive) with zeros. This command is similar to the
text editor ZERO command. Note that a block containing all
zeros is considered an empty block.

Example: 220 290 ZEROBLOCKS

will fill blocks 220 through 290 with zeros.

A	Switches to the <u>A</u> lternate block.
<line#> B	<u>B</u> egins entering lines.
<line#> BI	<u>B</u> egins <u>I</u> nserting lines.
<source-block#> <destination-block#> B-MOVE	Moves the specified block.
<block#> CHANGE	Changes the block#.
CLEAR	Clears the entire block, filling it with blanks.
<line#> D	<u>D</u> eletes the line.
<starting-line#> <ending-line#> DL	<u>D</u> eletes <u>L</u> ines.
<line#> E	<u>E</u> rases the line, filling it with blanks.
<block#> EDIT	Edits the specified block.
ERASE-CORE	Marks both core buffers as being empty.
FLUSH	Forces the writing onto secondary storage of both core buffers.
<line#> H	<u>H</u> olds the line in the text buffer.
<starting-line#> <ending-line#> HL	<u>H</u> olds <u>L</u> ines (for subsequent IL).
<line#> I	<u>I</u> nserts a line.
<line#> IL	<u>I</u> nserts <u>L</u> ines held by previous HL
L	<u>L</u> ists the block.
<line#> M	<u>M</u> odifies the line.
N	Switches to the <u>N</u> ext block.
<line#1> <line#2> O	<u>O</u> rs the two lines.
P	Switches to the <u>P</u> revious block.
<line#> R	<u>R</u> eplaces the line.
<line#> T	<u>T</u> ypes the line.
ZERO	Zeroes the entire block, filling it with zeroes.

Table 12.1 - Editor Commands

199 LOAD		Loads the block editor words.
<starting-block#> <ending-block#>	CLEARBLOCKS	Clears the blocks, filling each one with blanks.
<block#1> <block#2> <#-of-blocks>	EXCHANGE	Exchanges (swaps) the specified blocks.
<source-block#> <destination-block#>	INSERT	Inserts the specified block.
<source-block#> <destination-block#> <#-of-blocks>	MOVEBLOCKS	Moves the specified blocks.
<starting-block#> <ending-block#>	ZEROBLOCKS	Zeroes the blocks, filling each one with zeroes.

Table 12.1 - Block Editor Commands

13. PROGRAM STRUCTURE

Knowing that we may store our program code in a block on secondary storage, we are now ready to write some longer programs, utilizing this feature.

13.1 BLOCK ORIENTED PROGRAMS

If you have a group of definitions in a disc block you must `LOAD` the block in order to enter the definitions into the dictionary. This is accomplished by executing the sequence

`<block#> LOAD`

where `<block#>` is a single-word integer value specifying which block of secondary storage is to be loaded. The loading sequence starts with the first character in the specified block and continues through the block until either of the following words is encountered:

- `;S` - terminates loading of the current block and all characters that follow the `;S` in the block are ignored.
- `CONTINUED` - terminates loading of the current block but continues on to load the block whose block # is on top of the stack. All characters that follow the `CONTINUED` in the block are ignored.
- `-->` - terminates loading of the current block and continues on to load the next block. All characters that follow `-->` in the block are ignored. Refer to the end of this section for more detail concerning the use of `-->`.

Once again, some examples are the easiest way to see what is happening.

INPUT SEQUENCEBLOCK CONTENTSBLOCKS LOADED

250 LOAD

Block 250

STOP ;S

250

300 LOAD

301 LOAD

302 LOAD

Block 300

STOP ;S

Block 301

STOP ;S

Block 302

STOP ;S

300 ,

301 ,

302

300 LOAD

Block 300

301 CONTINUED

Block 301

302 CONTINUED

Block 302

STOP ;S

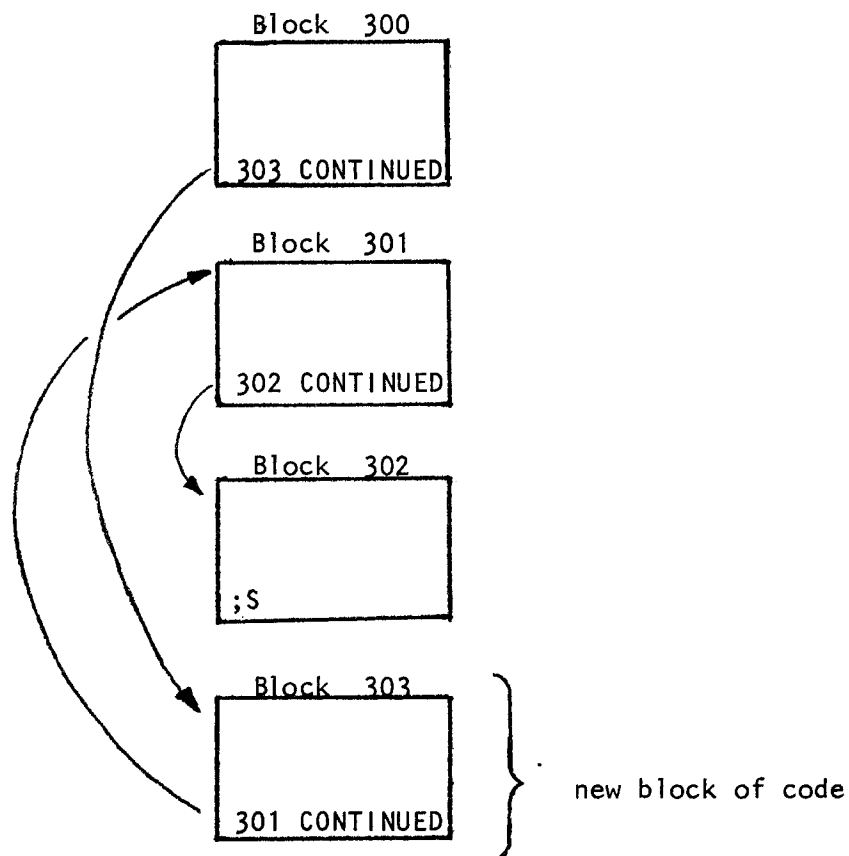
300 ,

301 ,

302

Notice how the last two examples perform identical functions, that is to load blocks 300, 301 and 302. The final example (using CONTINUED) is preferable since it requires only one LOAD to be keyed in and executed.

Now assume in the final example you wish to insert some code between blocks 300 and 301 (that is, the words to be entered into the dictionary must logically be loaded after block 300 has been loaded but before block 301 is loaded). One method to accomplish this is as follows:



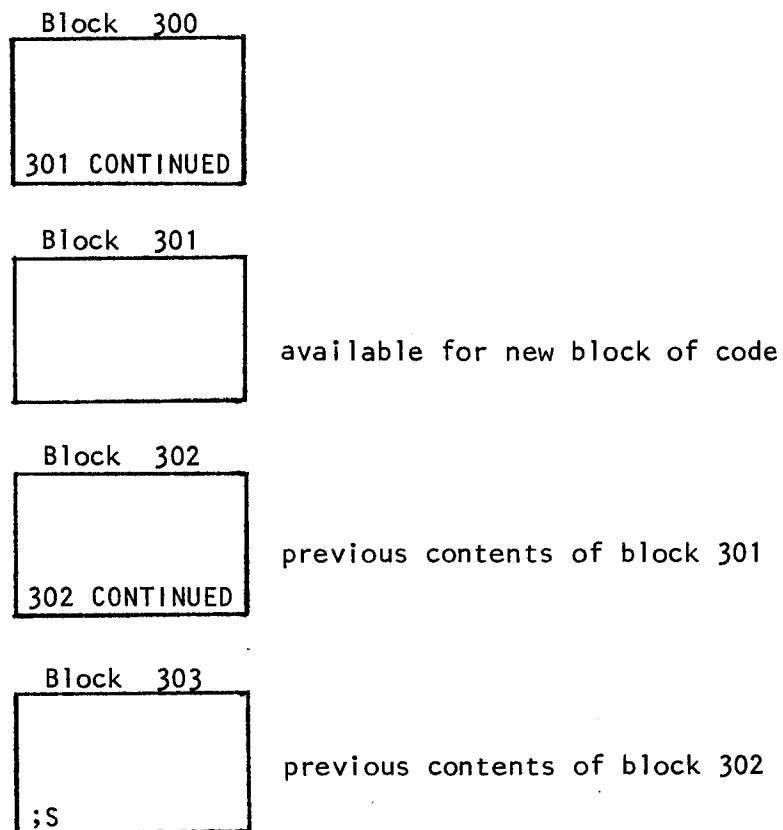
This method will work perfectly well, however the problem now is that the new code in block 303 is logically out of sequence since it should really appear between blocks 300 and 301. A few additions of this form and the program quickly becomes a mess!

To avoid this problem lets move the contents of block 302 to block 303 and then move the contents of block 301 to block 302. This then frees block 301 for the new code. We can use the block editor to move blocks 301 and 302 by executing

```
302 303 B-MOVE
```

```
301 302 B-MOVE
```

which will move the two blocks as desired. What we now have is:



Note however, that in addition to entering the new code into block 301, we must also edit the '302 CONTINUED' in block 302 to '301 CONTINUED' since we moved block 301 into block 302.

Finally, as a solution to this problem (the problem of having to edit the block# preceding CONTINUED), we can describe a word named +BLOCK which calculates an absolute block number given a relative block number:

<u>In block#</u>	<u>The sequence</u>	<u>Is identical to</u>
300	1 +BLOCK CONTINUED	301 CONTINUED
420	2 +BLOCK CONTINUED	422 CONTINUED
500	-3 +BLOCK CONTINUED	497 CONTINUED
300	-1 +BLOCK CONTINUED	299 CONTINUED

As the examples show, the single-word integer value preceding the word +BLOCK is added to the *current* block number yielding a single-word integer value absolute block number. (A block number such as 300, 420, etc. is called *absolute* since it refers to a specific block - a block number is called *relative* if it refers to a block in a specific location relative to the current block). The sequence

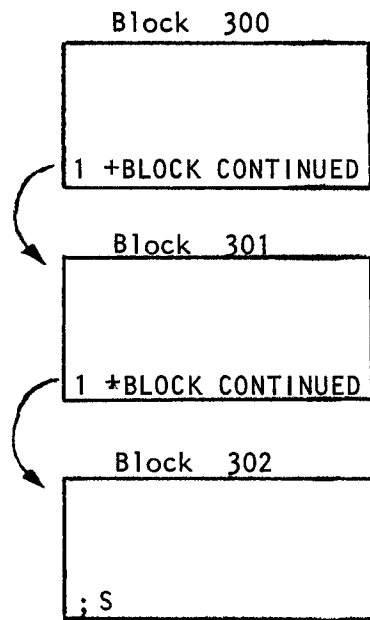
301 CONTINUED

is an absolute block reference since it refers to block 301 specifically. The sequence

1 +BLOCK CONTINUED

is a relative block reference since the block referred to depends on which block the sequence '1 +BLOCK CONTINUED' appears. The sequence '1 +BLOCK CONTINUED', if found in block 350, refers to absolute block 351; however if found in block 407, refers to absolute block 408.

Had a relative block reference been used in blocks 300 and 301 of the example we would not have to edit the '302 CONTINUED' as mentioned previously. Thus, the original appearance of the example should have been



After moving block 302 into 303 and block 301 into 302 we need only edit in the new code into block 301 and then terminate block 301 with

1 +BLOCK CONTINUED

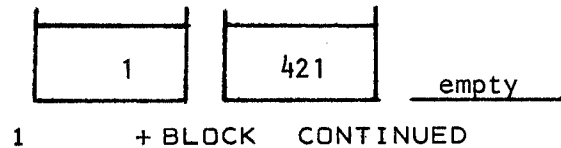
and we're all set. No other blocks need be modified.

The purpose of this lengthy discussion has been to explain the logic and usefulness behind the provided block structure. It would have been shorter to simply describe the +BLOCK word at the beginning, however it is more instructive to first examine the alternatives and see just why the final technique is the best. This discovery, examination and comparison of alternative techniques is the essence of programming.

The stack interactions of the word `+BLOCK` are very simple. Again consider the sequence

```
1 +BLOCK CONTINUED
```

being executed in block 420:



The word `+BLOCK` simply adds the single-word integer on top of the stack to the current block number, leaving the result on top of the stack.

Now that we know how to link together a sequence of blocks into a program what are the contents of the blocks? The format of each program block is largely (if not entirely) a matter of individual preference. The following format is used by the author and should be regarded as one possible format:

- 1) Line 1 contains the base in which all numeric quantities are specified (decimal or octal, specified by the FORTH words `DECIMAL` or `OCTAL`). *Never* assume that the base is set to what you want!
- 2) Following the base specification, the remainder of Line 1 is used to contain a comment which briefly describes the function of the code appearing in the block. A comment in FORTH is denoted by a left paren, followed by one or more spaces, followed by the comment, terminated by a right paren (this closing paren need not be preceded by a space). Thus to comment that a block contains hour angle calculations, the FORTH sequence would be

```
( HOUR ANGLE CALCULATIONS )
```

- 3) Lines 2 through 15 contain the code.

- 4) Line 16 contains the sequence

```
1  +BLOCK  CONTINUED
```

if there are more program blocks to follow, otherwise it contains the word

```
;S
```

to terminate the loading.

- 5) At the right margin of Line 16 one should put the date on which the block was last modified and the initials of the person making the modifications. This allows another person to see when the block was last changed and by whom. Note that since these characters appear after the CONTINUED or the ;S they are ignored during loading and therefore are effectively comments. For example, if the block was last modified by the author on February 30, 1976 one could write

```
30FEB76 WRS
```

to specify the date and the person.

Now consider a sample block, say block 503, as listed by the editor:

```
1.  DECIMAL                ( HOUR ANGLE CALCULATIONS )
2.
3.
.
.
.
15.
16.  1  +BLOCK  CONTINUED                                30FEB76 WRS
```

(Lines 2 through 15 would contain the program code.)

Summing up the recommended program structure:

- 1) Have each block load its successor with a relative CONTINUED, that is, for example

```
1  +BLOCK  CONTINUED
```

- 2) Terminate the final block only with ;S
- 3) The entire program is then loaded by executing

```
<starting-block#>  LOAD
```

where <starting-block#> is the absolute block number of the first program block.

As an alternative to specifying the absolute block number of the first block in a program, one can define a new word to be entered into the dictionary such that when this word is executed a specified block will be loaded. The word `LOADER` is used to define the new word and the format of this definition is

```
<block#>  LOADER  <name>
```

where <name> is the user assigned name (recognized as usual, by its first three characters and count). For example, instead of executing

```
300  LOAD
```

in the previous example, to load blocks 300, 301, 302 and 303 we can define the word `PROG` as follows

```
300  LOADER  PROG
```

and then simply execute the word `PROG` to load the four blocks. This technique has the advantage that one now remembers a user defined name, instead of a (possibly meaningless) block number.

The word `LOAD` starts loading with the first character of line 1 of a block and if one wishes to start loading with the first character of any line of a block, the word `LINELOAD` is available:

`<line#> <block#> LINELOAD`

As with the word `LOAD`, the loading of a block with `LINELOAD` terminates when `;S CONTINUED` or `-->` is encountered.

One should understand that the word `-->` is equivalent to the sequence

`1 +BLOCK CONTINUED`

The difference is that `-->` is a compiler directive which will be executed during the compilation of a colon definition. This means that one is able to extend a colon definition from one block to the next by placing the word `-->` as the final word in a block. This is not too frequent a requirement (if you write a colon definition that extends over a block, perhaps the definition should be broken into two or more words) however the facility is provided. In general, `-->` is preferred to `'1 +BLOCK CONTINUED'` because it is a single word (the latter is three words, each which must be executed) plus it requires fewer characters to be keyed in (hence less chance for typing mistakes). The sequence `'<value> +BLOCK CONTINUED'` should be used mainly when `<value>` is not equal to one.

13.2 FILE SYSTEM

A rudimentary file system is provided by KPNO Varian FORTH which allows one to allocate disc blocks into files. Use of the file system simplifies the combining of separate programs into a larger program by not forcing one to think in terms of absolute disc blocks.

A file is a sequential group of one or more disc blocks and every file has a name which is identified by its first three characters and count. A directory of all files is maintained on disc and for every file on disc, the directory contains the file name, absolute starting block number, and absolute ending block number (actually the ending block number plus one). As in FORTH, the file directory is searched backwards, therefore, if two files having the same name are both in the directory, only the newer file is accessible. A group of commands to manipulate files are provided by a file called FILEMAN (the file manager).

To load a file, one executes:

```
LOADFILE <name>
```

which LOADS the first block of the file. A file may LOADFILE other files and this nesting may go to a level of ten. All blocks within a file are numbered relative to the first block of a file, starting with zero. For example, a 4 block file would consist of blocks 0, 1, 2, and 3. In order to load block 3 from block 1 one could use

```
2 +BLOCK LOAD
```

Absolute block numbers must never be used in a file (unless one is referencing basic FORTH which will always reside in absolute blocks 8-199). The last line of block 0 of every file should contain a comment describing the file and the first 40 characters of this comment are printed by various file manager utilities in order to provide more information on the contents of a file than the file name alone provides.

In order to edit a file, one first executes

```
EDIT
```

which loads the editor words into the dictionary, followed by

```
FILE <name>
```

to specify which file is to be edited.

Finally

```
<relative-block#> EDIT
```

will edit any block within the file. If one then wishes to edit some other file simply execute

```
FILE <name>
```

```
<relative-block#> EDIT
```

In order to print a file on the lineprinter, one first executes

```
LOADFILE FPRINT
```

which loads the lineprinter words and a modified BLOCKPRINT utility into the dictionary. -FPRINT discards these words.

To print all non-zero blocks of a file, one executes

```
FPR <name>
```

To specify only certain blocks of a file to be printed, one executes

```
FILE <name>
```

```
m n BLOCKPRINT
```

which will print the relative blocks m through n inclusive of the specified file.

To print another file, simply re-execute either the FPR command or the FILE and BLOCKPRINT commands with the new filename.

The file manager is loaded by executing

```
LOADFILE FILEMAN
```

and may be discarded by executing -FILEMAN. The following file handling words are available after loading FILEMAN :

FCHANGE <name1> <name2>

Rename the file <name1> as <name2>.

FCOMPARE <name>

<name> specifies an existing disc file and any file on a file storage tape is compared with the disc file. The operator is asked which file on tape is to be compared. The relative block numbers of any non-equal blocks are printed.

FCOPY <name>

Copy a sequence of blocks between the specified file and a standard FORTH tape. The operator is asked whether the source is tape or disc and the absolute starting block number on tape. The number of blocks transferred equals the length of the file. If the destination is disc, then the disc file is zeroed before the transfer starts.

<#blocks> FCREATE <name>

Create a new file of the specified length. There is no check made to see if a file of the same name (first three characters and count) already exists. The disc area assigned to the file is neither cleared nor zeroed, hence one must assume the initial contents to be garbage.

FDELETE <name>

Delete the specified file from the directory. The disc space used by the file may be reused later by another file of the same or smaller size. If the files on either side of this file are empty (i.e. - were previously FDELETED) then the areas are combined in order to give the largest possible empty region.

FDUMP <name>

Dump the specified file from disc onto a file storage tape. The file is appended onto the end of the file storage tape. Preceding the file on tape is a header block specifying the name and size of the file. The file may later be loaded onto disc by FLOAD or LOADALL.

<n> FEXTEND <name>

Extend the length of the specified file by n blocks. A new file is created with the following characteristics: it retains the original filename, its length equals the sum of its original length plus n, all blocks are zeroed, all blocks from the original file are then transferred to this new file. The original file is then designated as EMPTY. If there is insufficient space available for the new extended file, the message NO SPACE will be printed.

FFREE

Print the number of free unused blocks located after the directory.

FINITIALIZE

Initialize a file directory. All files except the file manager are deleted.

FLIMIT	Set the upper limit block number for file storage. The operator is asked for the upper block number.
FLIST	List the entire disc directory on the terminal. For each file in the directory the file name, starting and ending absolute block numbers, length (in blocks), and the first 40 characters of the last line of block 0 of the file are listed. Any unused regions (where one or more files have been FDELETED) are listed with a name of EMPTY. In order to stop the listing before the terminal screen fills up type Control-S type Control-Q to resume listing.
FLOAD <name>	<name> specifies an existing disc file, and any file on a file storage tape may be loaded from tape into the disc file. The operator is asked which file on tape is to be loaded. If the lengths of the files on disc and tape are not equal, the number of blocks moved equals the shorter length.
FMOVE <name>	Move a sequence of blocks from an absolute location on disc into the specified file. The starting absolute block number of the source blocks on disc is requested and the number of blocks transferred equals the length of the file.

FSQUEEZE	Eliminate all imbedded empty files within the directory, if any occur, by squeezing user-filled files into consecutive block locations. All free unused blocks now reside in contiguous locations following the user-filled files.
FWHERE <name>	Print the absolute starting and ending block numbers of the specified file.
FZERO <name>	Zero all blocks contained in the specified file.
DUMPALL	Dump all non-empty files from disc onto a file storage tape, and then compare each block on tape to its corresponding block on disc.
LOADALL	Load all files from a file storage tape onto disc. Each file header on tape is examined and if a file of the same name exists on the disc, the file is loaded from tape onto disc (overwriting the previous contents of the disc file). If a file of the same name does not exist on disc, the file is created and loaded from tape. If there is insufficient room on disc for a file, the file is skipped on the tape.
FILE <name> <relative block#> LIST	Specify a file to be listed and list the contents of the specified block on the terminal.

NEWTAPE

Initialize a new magnetic tape which is to be used for file storage.

SAVEFILES

Write all blocks, from block 0 to the last block of the last directory file, to a tape and compare these blocks with their corresponding blocks on disc. It allows one to change the comment in the tape header, and it prints the final record number and final block number written to the tape. A tape generated by this word is a "boot tape" which may be loaded onto any system and run (Chapter 3).

TLIST

Read a file storage tape from beginning to end, listing in order every file that is on the tape (similar format to FLIST). In order to stop the listing before the terminal screen fills up, type Control-S type Control-Q to resume listing.

13.3 OVERLAYS

FORTH applications for which core-memory space is at a premium and run-time reloading exacts prohibitive time overhead will benefit from the use of disc overlays. An overlay is a section of pre-compiled dictionary stored on disc, which may be loaded directly into a reserved section of dictionary (an 'overlay-area') in memory. Any number of overlays may be prepared for loading into a given overlay-area. Also, any number of overlay-areas may be defined, subject to memory limitation. Optional paging is available for overlays containing data to be modified and retained throughout overlay operations. To establish an overlay-area, use the sequence

```
<size> 0-DEFINE <area-name>
```

The total length in words of the overlay-area is specified by <size>. <area-name> becomes the identifier of the overlay-area, and is compiled as the first word in the overlay-area. As usual, only the first three characters and count identify <area-name>. At this point the dictionary pointer is advanced to the end of the overlay-area. Usual dictionary operations may proceed, however, words defined after this point will not be accessible from overlays compiled into the preceeding area.

To create a new overlay for any given overlay-area, simply enter the name of the area. Use of the user defined word

```
<area-name>
```

causes dictionary linkages to be switched into the named overlay-area. Words subsequently defined become part of the new overlay. The only definitions accessible within an overlay are those made within the overlay itself and those made prior to definition of the overlay-area. These may include words within previously defined overlay-areas. To assist one in adjusting the lengths of overlays to fall within the defined area, the word

```
?LEFT
```

will push onto the stack the number of memory cells still available in the overlay-area.

When the overlay is complete it must be transferred to disc. The sequence

```
<block#>' O-SAVE <overlay-name>
```

accomplishes this, while also resetting linkages into the common dictionary.

The <overlay-name> becomes an entry in the common dictionary which is used in a special manner to access the overlay. The <block#> is the first block at which the overlay will be stored. Overlays greater than 512 words in length will be written into sequential blocks as required, 512 words per block. Partial blocks are zero-filled. Alternatively, an overlay may be saved without specifying a particular block number for each overlay. The VARIABLE O-BLK may be set to the starting block of a general disk area for overlay storage. The sequence

```
A-SAVE <overlay-name>
```

works like O-SAVE, except that the block number is taken from the variable O-BLK. After the overlay is transferred to disc, O-BLK is updated to the next available block number automatically. If the allotted size of an overlay is exceeded, attempts to save it will result in the message

```
n O-FLDW <overlay-name> ?Q
```

where n is the number of words remaining. The overlay-area itself and all subsequent dictionary entries are deleted.

The words defined within an overlay are accessible only when that overlay is in memory. An overlay may be explicitly loaded into core with the structure:

```
<overlay-name> O-LOAD
```

However, for operations making extensive use of words defined within overlays, it may be easier to use implicit overlay-loading. The sequence

```
<overlay-name> INCLUDES <word>
```

creates an external entry which automatically insures that <word> is available when required. <word> must be the identifier of a word already defined in the named overlay. Except for timing considerations and recognition that the originally defined <word> is no longer directly accessible, subsequent references to <word> may be made as if it was part of the common dictionary in core memory. Note that the word O-LOAD is superfluous to this scheme; it need never be used. The word INCLUDES insures that the proper overlay is loaded for compilation as well as for execution.

A typical usage of the overlays would be as follows:

```
- definitions for the common dictionary -
1024 O-DEFINE OVAREA    (1024 words = 2 blocks/overlay)
20  +BLOCK O-BLK !      (starting block# on disc)
OVAREA
- definitions to go into overlay #1 -
A-SAVE 10V
OVAREA
- definitions to go into overlay #2 -
A-SAVE 20V
OVAREA
- definitions to go into overlay #3 -
A-SAVE 30V
- definitions for the common dictionary -
```

In this example the overlay 10V will be stored in the first two blocks of the overlay region on disc, 20V in the next two blocks and 30V in the next two. In order to execute some words defined within 20V one must first execute

```
20V O-LOAD
```

Normally, variables or data defined within an overlay are reset to their initial states each time the overlay is reloaded into core memory. However, a simple scheme has been implemented which allows variables to survive the overlay reloading process. After an overlay has been defined and saved, setting the precedence bit of its area-name using the sequence

```
IMP <area-name>
```

will define it as a variable overlay. The precedence bit for a variable overlay can be properly set only when the following three conditions are met:

1. The overlay has already been placed on disc via
O-SAVE or A-SAVE.
2. The designated overlay is currently loaded into its core overlay-area.
3. Subsequent use of the overlay-area causes a different overlay to be loaded before any any dictionary is compiled into the area.

A brief description of the action of the overlay precedence bit may prove helpful: This bit is examined before a new overlay is brought into memory; if set, the current overlay is first re-written on disc. The precedence

bit is not examined before new dictionary is compiled into an overlay; consequently a variable overlay that happens to be loaded prior to compilation is not re-written on disc. The precedence bit of an overlay-area is cleared by either `SAVE` operation and so cannot be set during compilation. Although the <area-name> not the <overlay-name> is used to define a variable overlay, only the specifically designated overlay is affected. Remember, `IMP` is a toggle function. Once set the precedence bit will remain set until explicitly reset by another `IMP`.

Some helpful hints concerning the use of overlays:

The useful length of an overlay area is 9 words less than the total size of the area.

Do not attempt to write overlays past absolute block 2447.

Exceedingly long names or numeric strings (greater than 15 characters) entered at or near the end of the overlay-area could (conceivably, but unlikely) extend into the common dictionary during compilation and result in an `O-FLDW` error.

Do not attempt to `FORGET` an overlay-area name! The `FORGET` word does not handle overlay linkages properly. You may, however, `FORGET` any word ahead of an overlay-area and so delete the overlay-area as well.

Names of words which are not executed from outside an overlay may be freely duplicated in other overlays. However, it is inadvisable to duplicate names which are to be externally referenced.

Words defined in one overlay may be referenced from within a subsequently defined overlay. This could be a powerful tool, but must be used with care: If an overlay contains any words referenced by subsequent overlays, any modifications of it must be followed by re-compilation of the referencing overlays.

Don't forget that a small overlay defined for a large overlay-area requires the full disc space of the larger area.

Explicit or implicit loading of an overlay from disc does not actually result in a disc transfer if the requested overlay is already core-resident.

13.4 VOCABULARIES

A vocabulary is a logical subset of the dictionary. Basic FORTH includes three vocabularies:

FORTH - the set of words comprising basic FORTH;

ASSEMBLER - the set of words which create machine code;

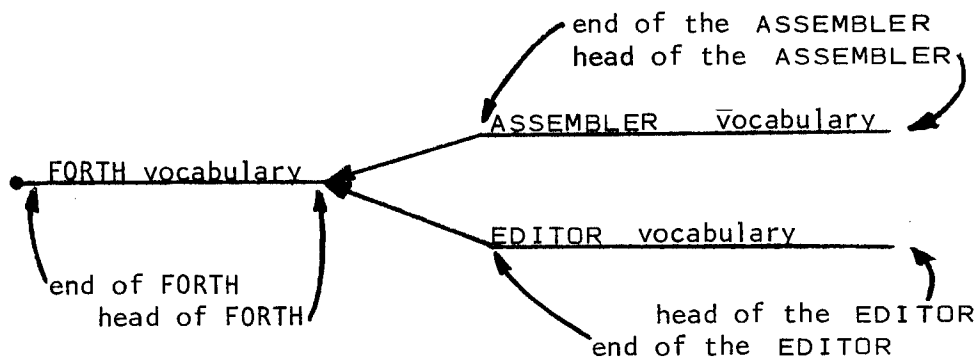
EDITOR - the set of words which create and modify source blocks.

A vocabulary may be used in two distinct ways:

- 1) A place to look for existing words in the dictionary
- dictionary searches begin in the CONTEXT vocabulary.

- 2) A place to put new words into the dictionary - new words are placed in the CURRENT vocabulary.

The end of one vocabulary may point to another vocabulary and this is referred to a "chaining". Normally, both the ASSEMBLER vocabulary and the EDITOR vocabulary are chained to the FORTH vocabulary. The logical structure is then



direction of dictionary searches

The head of a vocabulary is the *last* word entered into that vocabulary and the end is the *first* word entered (recall that all dictionary searches in FORTH start at the most recent word and proceed backwards). It is important to note that the structure shown above is the "logical" structure of the dictionary - the words that comprise any given vocabulary need not reside in sequential dictionary locations. Every dictionary entry has a link that points to the previous word in the vocabulary and this link field converts the physical dictionary structure into its logical vocabulary tree.

To specify which vocabulary is to be searched for existing entries (the `CONTEXT` vocabulary) simply execute the name of the vocabulary. Executing `FORTH` will cause only the `FORTH` vocabulary to be searched. Executing `ASSEMBLER` causes the `ASSEMBLER` vocabulary to be searched first and if the word is not found then the search automatically continues on to the `FORTH` vocabulary (since `ASSEMBLER` is chained to `FORTH`). In this case the `EDITOR` vocabulary is not searched. Similarly, if `EDITOR` is executed then the `EDITOR` vocabulary followed by the `FORTH` vocabulary will be searched. The `ASSEMBLER` vocabulary is not searched.

To specify which vocabulary is to receive new definitions (the `CURRENT` vocabulary) execute

```
<name> DEFINITIONS
```

where `<name>` is the name of a previously defined vocabulary. A new vocabulary is defined by executing

```
VOCABULARY <name>
```

The dictionary entry for `<name>` is entered into whatever vocabulary is currently receiving definitions, *not* into the new vocabulary being created. Therefore, to keep newly defined vocabularies generally accessible, the sequence

```
FORTH DEFINITIONS
```

should precede any new vocabulary definition.

To facilitate vocabulary control in basic `FORTH`, automatic context switching has been built into some basic words. The `EDITOR` vocabulary is automatically selected by the sequence '`<block#> EDIT`'. The words that begin machine code definitions (`CODE`, `SUBROUTINE`, `;CODE`, `ORC.x` and `!CODE`) automatically execute `ASSEMBLER` to start dictionary searching in the `ASSEMBLER` vocabulary. The words that begin a colon definition (`:`, `:ORX` and `!:`) automatically force dictionary searching to begin in the same vocabulary that will receive the new definition.

The sequence

```
<name> VLIST
```

may be executed to list all the entries in a given vocabulary. Note that if the vocabulary is chained to another vocabulary then the listing will automatically continue with the chained vocabulary.

Each Vocabulary name is a compiler directive; that is, a word which is automatically executed when it appears within a colon-definition. This feature allows access to specialized vocabularies by simply including the Vocabulary-name within a definition. For example, a colon-definition being compiled into Vocabulary V3 needs a word that was defined in V2. Within the definition, the sequence,

```
: . . . V2 <word> V3 . . . ;
```

performs the requisite switching, then returns to the current vocabulary. Only the address of <word> is actually compiled.

A unique variable must be provided to keep track of the last word in each vocabulary. This special "head" is created as part of each Vocabulary-name and is updated as new words are added to the Vocabulary. It is accessed (indirectly) through the VARIABLES CURRENT and CONTEXT:

Dictionary searches begin in the "context" vocabulary: Executing a Vocabulary-name makes CONTEXT point to the unique "head" for that Vocabulary.

New words are placed in the "current" vocabulary: Executing the word DEFINITIONS sets CURRENT, making new words go into the last-named Vocabulary.

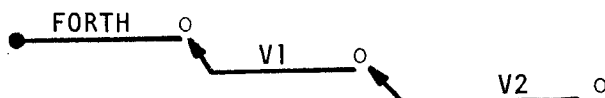
A vocabulary normally has access only to those words available when it was defined. By way of example, consider the sequence:

```
FORTH DEFINITIONS
```

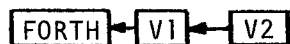
```
VOCABULARY V1 V1 DEFINITIONS <words for V1>
```

```
VOCABULARY V2 V2 DEFINITIONS <words for V2>
```

which results in a logical structure like:



and a physical dictionary structure:

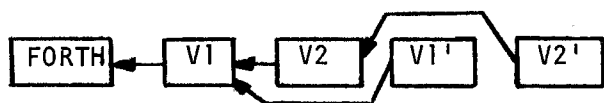
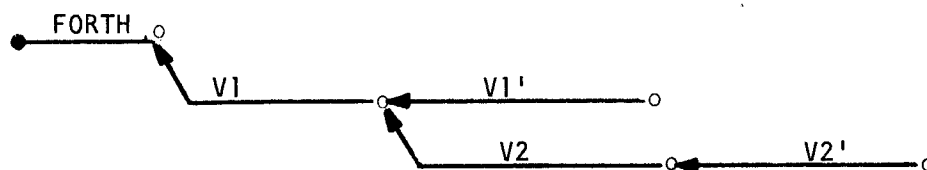


Definitions subsequently added to V1 are inaccessible to subsequent additions in V2. A continuation of the above sequence:

```
V1 DEFINITIONS <words for V1'>
```

```
V2 DEFINITIONS <words for V2'>
```

creates extensions to the original vocabularies that we shall refer to as "dialects".



As illustrated in the diagram of the resulting structures, Dialect V2' does not have access to Dialect V1'.

The word `CHAIN` may be used to couple one vocabulary to future extensions of another. In the example, the sequence:

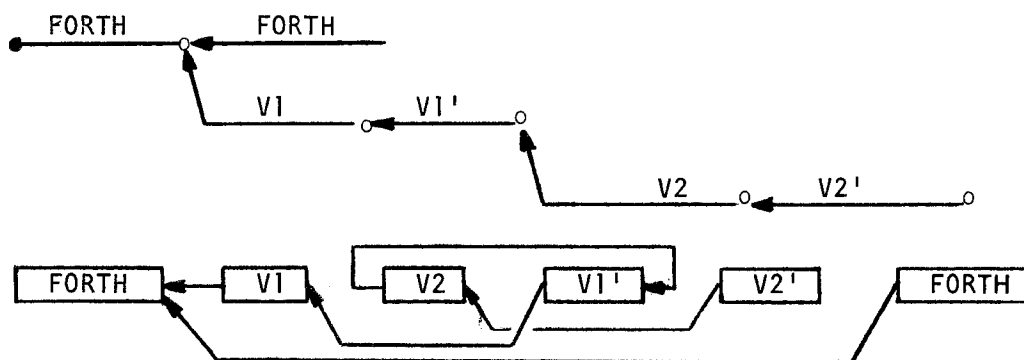
```
V2 DEFINITIONS CHAIN V1
```

will modify the links between vocabularies such that all words subsequently defined in V1 will be available to V2.

Note that since V1 is not chained to FORTH, if we subsequently create more

```
FORTH DEFINITIONS ... ..
```

the resultant structure:



does not allow either dialect of V1 or V2 to access the new FORTH' definitions.

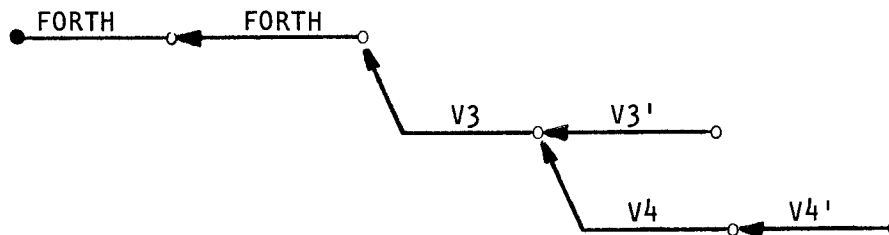
A variation on the preceding vocabulary-creation sequences illustrates the flexibility of chaining techniques. We create new vocabularies as before, but with the first one chained to FORTH, thusly:

```
FORTH DEFINITIONS
VOCABULARY V3 V3 DEFINITIONS CHAIN FORTH <words for V3>
VOCABULARY V4 V4 DEFINITIONS <words for V4>
```

We then define additional dialects:

```
FORTH DEFINITIONS <words for FORTH'>
V3 DEFINITIONS <words for V3'>
V4 DEFINITIONS <words for V4'>
```

thereby creating a vocabulary structure:



which allows all V3 and V4 dialects access to all of FORTH while isolating the words in dialect V3' from vocabulary V4.

14. TERMINAL I/O

Terminal I/O involves the input and output of both numbers and character strings between the program and the operator. The output of character strings, the input of numbers and the output of numbers are each described separately. The input of character strings is somewhat complicated and is not discussed in this primer.

14.1 CHARACTER OUTPUT

The output of a character string is accomplished by preceding the string with the two character sequence `."` and then terminating the string with a quote mark. The sequence `."` is a FORTH word (Section 4.1) and must therefore be followed by a space. The final quote need not be preceded by a space. Consider the following examples:

<u>FORTH Character String</u>	<u>Will be Printed as</u>	<u># of Characters</u>
<code>."</code> HELLO <code>"</code>	HELLO	5
<code>."</code> HELLO <code>"</code>	HELLO	6
<code>."</code> START MOTOR <code>"</code>	START MOTOR	11
<code>."</code> X <code>"</code> <code>."</code> Y <code>"</code>	XY	1, 1
<code>."</code> X <code>"</code> <code>."</code> Y <code>"</code>	X Y	1, 2

The important point to note in these examples is that any spaces in addition to the one required space (that follows the `."`) are considered as part of the character string.

Special characters (i.e. - the non-printing characters such as carriage-return, line-feed, bell, etc.) may also be included in the character string. A list of all special characters available in the ASCII character set is given in Appendix A. One frequent use of these special characters is to include a BELL character (Control-G) in a string that is to be printed on a CRT terminal to alert the operator that a message has been printed.

To assist in the formatting of character strings for output three additional words are defined in FORTH:

SPACE - will print a single space.
<spaces> SPACES - will print a string of spaces.
CR - will print a carriage-return.

For example, the sequence

```
CR ." XY = 2" CR SPACE ." Z = 5"
```

will print XY = 2
 Z = 5

↑
└─ column 1 of terminal

All character output described in this Section can be executed as shown or placed within a colon definition. For example, the word PRN, defined as

```
: PRN CR ." XY = 2" CR SPACE ." Z = 5" ;
```

and when executed will produce the same output as shown above.

14.2 NUMERIC INPUT

Numeric input involves the reading in of numbers and the conversion into the appropriate data type (single-word integer, double-word integer or floating-point number). FORTH provides three words to accomplish this input:

SASK	Reads a single-word integer and pushes its value onto the stack. The number must <i>not</i> contain a decimal point.
DASK	Reads a double-word integer and pushes its value onto the stack. The number <i>must</i> contain a comma.
FASK	Reads a floating-point number and pushes its value onto the stack. The number <i>must</i> contain a decimal point.

If there is an error in the number entered (for example a non-digit detected) the message

```
?  RETRY  #
```

will be output and FORTH will wait for the number to be re-entered. Additionally, if the error is due to either a single-word integer being entered with a decimal point or a double-word integer or floating-point number being entered without a decimal point, one of the following messages will be output:

```
ND  .  ALLOWED
#    MUST CONTAIN  .
```

These three asking words do *not* inform the operator that they are waiting for a number to be input - this must be done by the program. For example, consider the word ?TEMP that inputs a temperature:

```
0  VARIABLE  TEMP
:  ?TEMP  ."  ENTER  TEMP  (DEG.  C)"  SASK  TEMP  !  CR  ;
```

There must be no digits to the right of the comma in a double-word integer while a floating-point number will usually contain digits to the right of the decimal point. Floating-point numbers input by the word FASK may be entered as a fraction to a given power, for example 0.5E2 is the same as 50.0.

14.3 NUMERIC OUTPUT

Numeric output involves the printing of a number on the stack in some specified format. We have already encountered three words used for this purpose: `.` (to print single-word integers), `D.` (to print double-word integers) and `F.` (to print floating-point numbers). This section will expand on these words to provide additional output capabilities.

To summarize the words that we will describe in this section:

<u>sequence</u>	<u>outputs</u>
<value> <code>.</code>	single-word integer, free format
<address> <code>?</code>	single-word integer, free format
<value> <code>B.</code>	single-word integer, 16 binary digits
<DW-value> <code>D.</code>	double-word integer
<FP-value> <code>E.</code>	floating-point with exponent
<address> <code>E?</code>	floating-point with exponent
<FP-value> <code>F.</code>	floating-point without exponent
<address> <code>F?</code>	floating-point without exponent
<FP-value> <code>G.</code>	generalized floating-point
<address> <code>G?</code>	generalized floating-point
<value> <code>H.</code>	single-word integer, hexadecimal
<value> <field> <code>I.</code>	single-word integer
<value> <field> <#places> <code>N.</code>	single-word integer
<value> <code>O.</code>	single-word integer, 6 octal digits
<value> <code>S.</code>	single-word integer
<value> <code>U.</code>	single-word integer, unsigned

Note - as with all FORTH words, these words, after printing the number on top of the stack, will pop the number from the stack.

The word `.` is the simplest - the single-word integer value on top of the stack is printed using the minimum field width required. The number is first preceded by a space. For example, the integer 25 requires a field width of two positions and the integer -25 requires a field width of three positions.

In order to gain more control over the printing of single-word integers, the word `S.` must be used. The user must set the `VARIABLE FLD` to the minimum field width desired. If the number requires more positions than specified, then the number will exceed the specified field width. There is no preceding space printed by this word. If `FLD` is set to 0 then free format is used (minimum field width). Additionally, the `VARIABLE DPL` controls the number of digits to be printed to the right of the decimal point. A value of -1 specifies that the decimal point is not to be printed. Executing the word `FREE` will set `FLD` to 0 and `DPL` to -1 (minimum field width and no decimal point). The word `FREE` is defined as

```
0 FLD -1 DPL 2SET FREE
```

and the definition of `.` then becomes

```
: . FREE SPACE S. ;
```

The sequence

```
3 FLD ! (SET FIELD WIDTH TO 3) -1 DPL !
CR 5 S. CR 29 S. CR -58 S. CR 2.09 S. CR
```

will output

```
5
29
-58
2109
└─ column 1 of terminal
```

Also, the sequence

```
2 FLD ! -1 DPL !
5 S. 1 S. 28 S.
```

will output `5 128` since there is no space preceding each number.

The sequence

```
5 . 1 . 28 .
```

will output `5 1 28` since each number is preceded by a space.

Note that when outputting a sequence of numbers `FLD` and `DPL` need only be set once and will remain in effect until modified again.

The words `O.` `B.` and `H.` are used to print a single-word integer in octal, binary and hexadecimal respectively. The sequence

<value> <field-width> I.

allows one to print a single-word integer and specify the <field-width> on the stack. If one wishes to print a single-word integer with a decimal point, the sequence

<value> <field-width> <#places> N.

may be used, where <#places> is the number of digits to be printed to the right of the decimal point. The words I. and N. are defined in FORTH as

```
: N. DPL ! FLD ! S. ;  
: I. -1 N. ;
```

The word U. is used to print a single-word integer as an unsigned 16-bit number.

The word D. is similar to S. and is used to output double-word integers. The user must set the VARIABLE FLD to the minimum field width desired and if the number requires more positions than specified then it will exceed the field width. There is no preceding space printed by this word. As with S. the VARIABLE DPL specifies the number of digits to be printed to the right of the decimal point (a value of -1 specifies no decimal point). The sequence

```
7 FLD ! ( SET FIELD WIDTH TO 7 ) 0 DPL !  
CR 32768, D. CR -65000, D. CR 1234567, D. CR
```

will output 32768.

-65000.

1234567.

↑
└─ column 1 of terminal

The decimal point that may be printed must be included in the specified field width (i.e. - the double-word integer -1, requires a minimum field width of 3).

```
25, DOUBLE X  
4 FLD ! 0 DPL !  
CR X D@ D. 1 DPL ! CR X D@ D.
```

will output 25.

2.5

↑
└─ column 1 of terminal

Floating-point numbers may be printed in one of two ways: with or without an exponent. The word F. prints a floating-point number without an exponent and the sequence

<field-width> <#digits-to-right-of-decimal-point> W.D

must be executed to set both the total field width (including decimal point) and the number of digits to be printed to the right of the decimal point.

For example, the sequence

8 3 W.D

specifies a field width of 8 with three digits to the right of the decimal point (similar to the Fortran F8.3 format). The sequence

8 3 W.D

CR 25.1 F. CR 3.14159 F.

will output 25.100

3.142

↑
column 1 of terminal

(Note the rounding that is automatically performed.) As with the words F. and D. the format control (by executing W.D) need only be specified once for a sequence of numbers.

The word E. is used to print floating-point numbers with an explicit exponent. The word W.D is used as above to control the printing of the fraction which is then followed by 'E', followed by a 3-digit exponent. For example,

8 3 W.D

3.14159 E.	will output	3.142E0
.314159 E.	will output	3.142E-1
31415.9 E.	will output	3.142E4

In floating-point output using either E. or F. the user must be careful not to print more than 9 significant digits -- doing so will cause an asterisk to be printed preceding the number, indicating overflow on conversion. This means that very small numbers and very large numbers must be output with E. and not with F..

The word G. will print a floating-point number using either E. or F., depending on the size of the number. If the number is either too large or too small for F. then E. will be used.

Note that the words ? E? F? and G? do not print the top number on the stack, rather they require the address of the number to be printed to be on top of the stack. For example, given the definition

```
0 VARIABLE A
0. REAL B
```

then

```
A @ .
B F@ F.
```

are equivalent to

```
A ?
B F?
```

One is simply combining the two words @ and . into the word ?. This combining of two or more words into a single word is discussed more thoroughly in Section 15.4.

EXERCISES - CHAPTER 14

- 1) Define a word named IMIN that reads a sequence of non-negative integers from the operator and prints the minimum value. The end of the input is signalled by a negative number being entered, at which point the minimum should be printed.

- 2) A Fibonacci number is a number in the infinite sequence
0, 1, 1, 2, 3, 5, 8, 13, 21, 34

where the first two terms are 0 and 1 and each successive term is the sum of the two preceding terms. Define a word named FIB that calculates and prints all Fibonacci numbers $\leq N$ where N is the integer on top of the stack. For example

20 FIB should print 0 1 1 2 3 5 8 13

15. ADVANCED ARITHMETIC

This chapter complements Chapters 5 and 7 by completing the description of the arithmetic words available in FORTH.

15.1 NUMERICAL FUNCTIONS

A table of the standard numerical functions provided by FORTH is given in Table 15.1. The operation of all these words is similar in that they all expect one or more parameters on the stack and then after popping the parameter(s) from the stack the result(s) are pushed onto the stack.

The naming convention of these arithmetic words is to prefix the name with a "D" or an "F" if the word operates on a double-word integer or a floating point number, respectively. The second character of a trigonometric word will be "D" if the mode is degrees or "R" if the mode is radians.

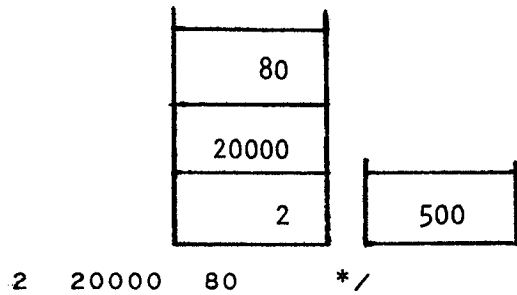
SINGLE-WORD INTEGER	DOUBLE-WORD INTEGER	FLOATING POINT	DESCRIPTION
/	D/		Multiplies the second number on the stack by the third number on the stack, forming a double-word integer temporary result. This temporary result is then divided by the top number on the stack leaving a single-word integer result on top of the stack.
MOD			Divides the second number on the stack by the top number on the stack, leaving the remainder on top of the stack. (See note 1)
/MOD			Divides the second number on the stack by the top number on the stack, leaving the quotient on top of the stack and the remainder below. (See note 1)
MINUS	DMINUS	FMINUS	Changes the sign of the top number on the stack.
ABS	DABS	FABS	Replaces the top number on the stack with its absolute value.
MAX	DMAX	FMAX	Replaces the top two numbers on the stack with the larger of the two numbers.
MIN	DMIN	FMIN	Replaces the top two numbers on the stack with the smaller of the two numbers.
SQRT		FSQRT	Replaces the top number on the stack with its square root (SQRT calculates the single-word integer truncated square root of a double-word integer).
		FDSIN FDCOS FDTAN	Replaces the top number on the stack (an angle in degrees) with either is sine, cosine or tangent.

SINGLE-WORD INTEGER	DOUBLE-WORD INTEGER	FLOATING POINT	DESCRIPTION
		FDATN	Divides the second number on the stack by the top number on the stack and then interprets the result as an angle (in degrees) and leaves its arctangent on top of the stack. (See note 2)
		F2XP	Replaces x by 2^x (where x is the top number on the stack).
		FEXP	Replaces x by e^x .
		FEXP10	Replaces x by 10^x .
		FLN	Replaces x by $\ln(x)$.
		F2LOG	Replaces x by $\log_2(x)$.
		FLOG	Replaces x by $\log_{10}(x)$.

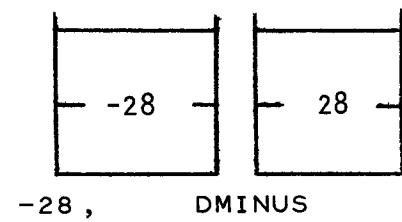
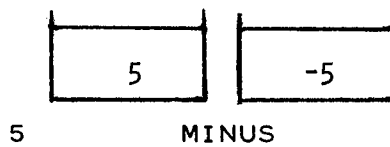
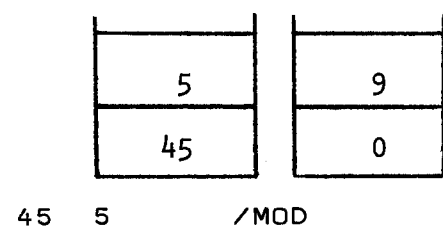
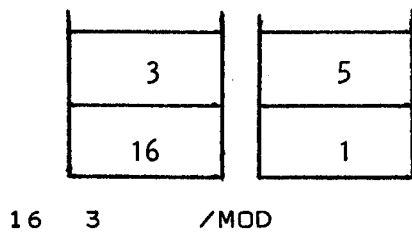
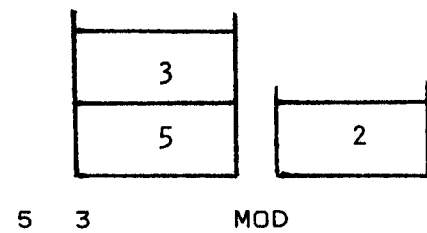
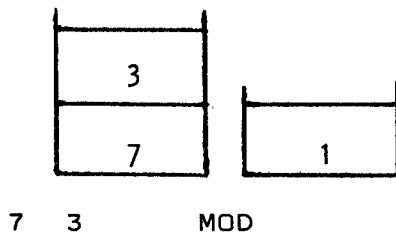
- Notes:
1. The sign of the remainder will be the sign of the dividend.
 2. The result will be in the range [0,360).

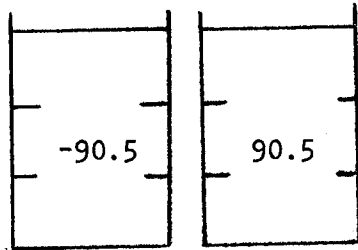
Table 5.1 - ARITHMETIC FUNCTIONS

The following examples should elucidate some of the descriptions given in Table 5.1.



(note that the intermediate product = 40,000 which requires the double-word intermediate result)

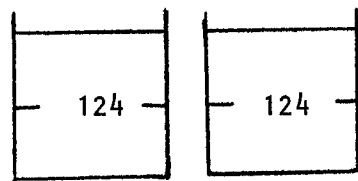




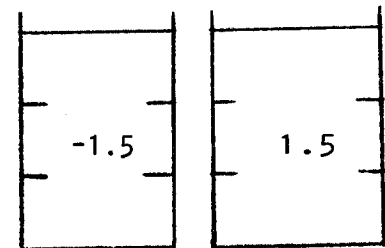
-90.5 FMINUS



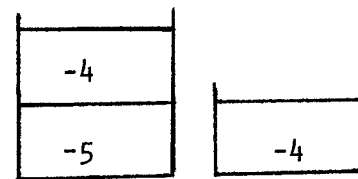
-100 ABS



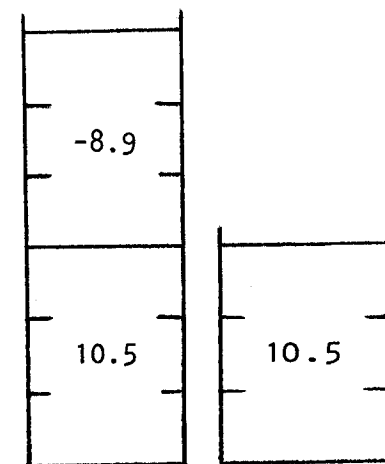
124 , DABS



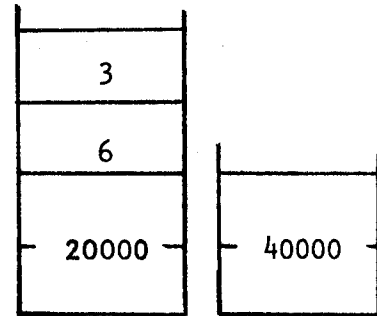
-1.5 FABS



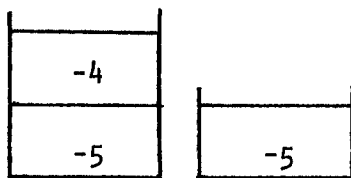
-5 -4 MAX



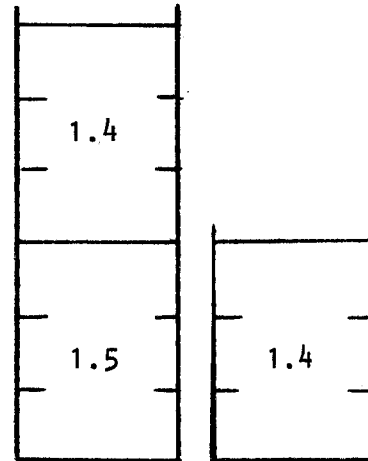
10.5 -8.9 FMAX



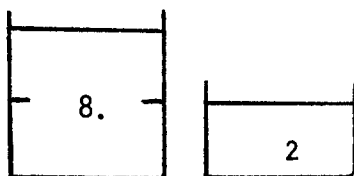
20000, 6 3 D*/



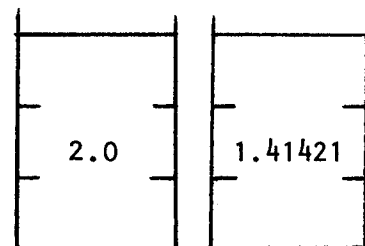
-5 -4 MIN



1.5 1.4 FMIN



8, SQRT

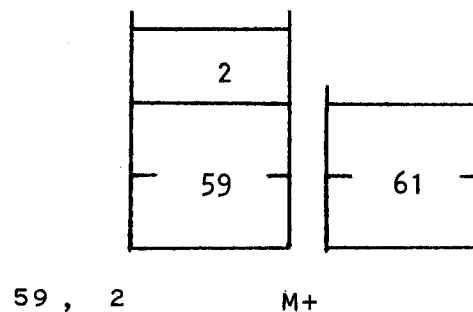


2.0 FSQRT

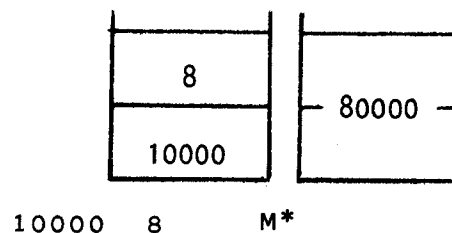
15.2 MIXED PRECISION OPERATORS

Mixed precision arithmetic operators perform arithmetic on numbers of different modes. In FORTH the mixed precision words usually operate on a single-word integer and a double-word integer. For example, the word `+` adds together two single-word integers, the word `D+` adds together two double-word integers and finally the mixed precision word `M+` adds a single-word integer to a double-word integer, leaving of course, a double-word result. Similar to the prefixing of double-word operators and floating point operators with `"D"` and `"F"`, the mixed precision operators are prefixed with an `"M"`.

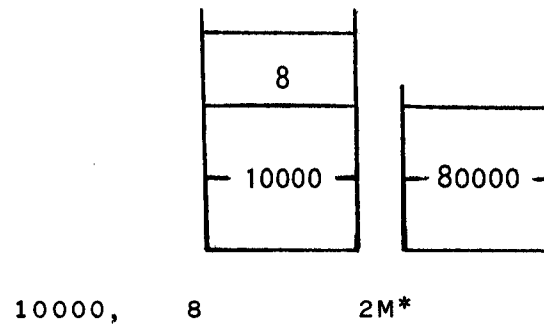
M+ Adds a single-word integer on top of the stack to the double-word integer below, leaving the double-word integer sum on top of the stack.



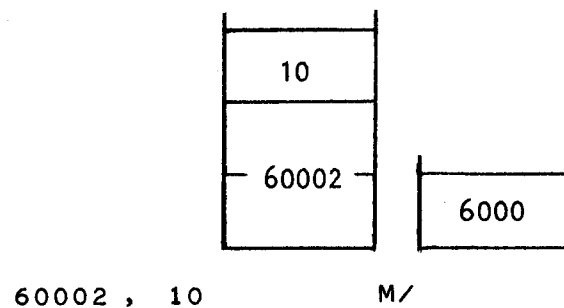
M* Multiplies the single-word integer on top of the stack by the single-word integer below, leaving the double-word integer product on top of the stack. (This word differs from the word `*` in that `*` calculates only a single-word integer product while `M*` calculates the full double-word integer product.)



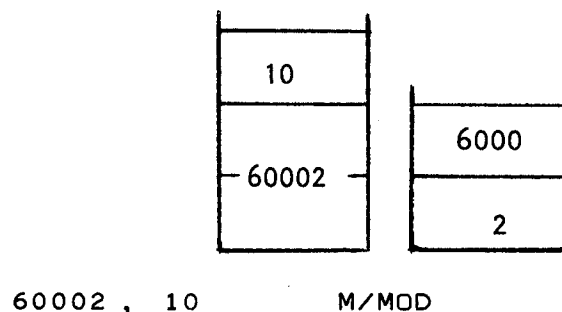
2M* Multiplies the single-word integer on top of the stack by the double-word integer below, giving a double-word integer result.



M/ Divides the double-word integer in the second position on the stack by the single-word integer on top of the stack, leaving the single-word integer quotient on top of the stack.



M/MOD Divides the double-word integer in the second position of the stack by the single-word integer on top of the stack, leaving the single-word integer quotient on top of the stack and the single-word integer remainder below.



15.3 ARITHMETIC RANGE ERRORS

In the implementation of KPN0 Varian FORTH it was decided to ignore all arithmetic range errors, that is overflow and underflow. Overflow occurs when the result of an operation requires more precision than is provided. For example, the largest positive single-word integer that may be represented is 32,767 (Section 7.1) hence the sequence

20000 20000 +

will generate an erroneous result since the result (40,000) is larger than 32,767. Similarly, the sequence

40000 , SFIX

will also produce an erroneous result. Overflow can also occur in floating-point operations if the range of the exponent exceeds the largest possible exponent. Underflow can also occur in floating point operations if the exponent is smaller than the smallest possible exponent. Fortunately in Varian FORTH the range of the exponent in a floating-point number is large enough so that exponent overflow and exponent underflow are extremely unlikely. However, standard arithmetic overflow is a likely possibility and it is up to the user to determine the probable range of his variables when writing a program and use the appropriate data structures so as to avoid overflow.

15.4 COMBINED WORDS

In the interest of core efficiency and execution speed, there exist many "combined" words which take two or more commonly used words (which are executed in sequence) and combine them into a single FORTH word.

For example, in Chapter 14 we saw where the sequence

@ .

was combined into the word ?. If the original sequence (@ . in this example) appears more than 5 times in one's program then the definition of the word ? and its use will reduce the amount of memory used by the program. This technique is the essence of FORTH - combining primitive operators into more general and useful words.

Some additional words which one is bound to encounter are:

1+	<u>equivalent to</u>	1 +
1-		1 -
2+		2 +
2*		2 *
2/		2 /
D2*		2 1 D*/
D2/		1 2 D*/
F2*		2.0 F*
F2/		2.0 F/
+!		+ !
F+!		F+ F!
FSQ		3DUP F*
1+!		1+ !
1-!		1- !

One could define these words as one would expect, for example

```
: 1+ 1 + ;
```

and this will work fine, however, many of the commonly used words listed above are coded directly in machine language thereby providing a decrease in execution time in addition to a decrease in memory requirements.

One should always use these combined words, rather than the original sequence, as the implementation of a particular FORTH system may take advantage of certain machine features in order to optimize the combined word. For example, in KPN0 Varian FORTH the words 2* 2/ D2* and D2/

are all implemented as arithmetic shifts. The words $F2^*$ and $F2/$ are implemented as an increment or decrement by one of the floating-point number's exponent. The word $1+!$ is implemented by a single machine instruction.

EXERCISES - CHAPTER 15

- 1) Define a word named ?ODD/EVEN that will test the integer on top of the stack and then print either "ODD" or "EVEN".
- 2) If the top word on the stack is a double-word integer representing the number of seconds past midnight, define a word named ?TIME that takes this value and prints the hour, minute and second.

Sample values are:

1,	--->	0 (hours)	0 (minutes)	1 (seconds)
60,	--->	0	1	0
3600,	--->	1	0	0
14399,	--->	3	59	59
30332,	--->	8	25	32

- 3) Define a word named QUAD that solves the quadratic equation

$$ax^2 + bx + c = 0$$

using the formula

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The parameters a, b & c will be the top three floating-point numbers on the stack (c on top, b below c and a below b). The output should indicate the type of solution (one real root, two real roots or two complex conjugate roots). For example

1.0 5.5 -10.5 QUAD	should print TWO REAL ROOTS: 1.5 -7.0
4.0 -292.0 5329.0 QUAD	should print ONE REAL ROOT: 36.5

3.0 1.0 5.0 QUAD

should print

TWO COMPLEX ROOTS (REAL &
COMPLEX PARTS):
-.16667 1.28019

- 4) Define a word named FDASINE that calculates the angle (in degrees) whose sine is the floating-point number on top of the stack. Use the equation

$$\begin{aligned}\arcsin(x) &= \arctan \left[\frac{x}{\sqrt{1-x^2}} \right] & |x| < 1 \\ &= 90^\circ & x = 1 \\ &= -90^\circ & x = -1\end{aligned}$$

For example

0.0	FDASINE	F.	should print	0.0
1.0	FDASINE	F.	should print	90.0
-0.5	FDASINE	F.	should print	-30.0

Be sure to handle the case of $x = \pm 1$.

- 5) Define a word named FDACOS that calculates the angle (in degrees) whose cosine is the floating-point number on top of the stack. Use the equation

$$\begin{aligned}\arccos(x) &= 180 - \arcsin(\sqrt{1-x^2}) & -1 \leq x \leq 0 \\ &= \arcsin(\sqrt{1-x^2}) & 0 \leq x \leq 1\end{aligned}$$

For example

0.0	FDACOS	F.	should print	90.0
1.0	FDACOS	F.	should print	0.0
-0.5	FDACOS	F.	should print	120.0

- 6) Define a word named `x**y` that raises a floating-point number `x` (second number on the stack) to the floating-point power `y` (top number on the stack). Use the formula

$$x^y = e^{y(\ln x)}$$

(where $e = 2.71828182$). Print an error message if $x < 0$ (since $\ln x$ is undefined for negative arguments). For example

5.0	2.0	<code>x**y</code>	<code>F.</code>	should print	25.0
2.0	12.0	<code>x**y</code>	<code>F.</code>	should print	4095.99922

- 7) Using the word `x**y` from the previous exercise, define a word named `FCUBERT` that calculates the cube root of the floating-point number on top of the stack. The argument may be negative in which case the final answer must then be negated (to avoid the error message from `x**y`). For example

64.0	<code>FCUBERT</code>	<code>F.</code>	should print	4.0
-8.0	<code>FCUBERT</code>	<code>F.</code>	should print	-2.0

- 8) Define a word named `FROUND` that will *round* a floating-point number prior to the floating-point number being truncated by either `SFIX` or `DFIX`. For example

4.4	<code>FROUND</code>	<code>SFIX</code>	<code>.</code>	should print	4
4.5	<code>FROUND</code>	<code>SFIX</code>	<code>.</code>	should print	5

- 9) Define a word named `I**J` that raises a single-word integer, `i` (second number on the stack) to a single-word integer power, `j` (top number on the stack). Use the formula given in exercise 6 above. For example

10	3	<code>I**J</code>	.	should print	1000
7	5	<code>I**J</code>	.	should print	16807

- 10) Define a word named `CUBE` that cubes the single-word integer on top of the stack.. For example

1	<code>CUBE</code>	.	should print	1
-5	<code>CUBE</code>	.	should print	-125

Use the word `I**J` from the previous exercise.

- 11) Instead of using the word `I**J` in forming the cube of a single-word integer (previous exercise) one could define `CUBE` as

```
: CUBE  DUP  DUP  *  *  ;
```

Which definition do you think is preferable and why?

- 12) An iterative algorithm is one that repeats itself (iterates) until a certain value is within a specified range. For example, an iterative algorithm to determine the square root (`y`) of a number (`x`) is

$$y_0 = .5903x + .4173$$

$$y_{i+1} = \frac{1}{2} \left(y_i + \frac{x}{y_i} \right) \quad i = 0, 1, 2, \dots$$

In this example the initial approximation (y_0) is calculated and then used to calculate y_1 . y_1 is then used to calculate y_2 , y_2 is then used to calculate y_3 , etc. The algorithm terminates when the value y converges to its limiting value. This is determined by

$$\frac{|y_{i+1} - y_i|}{|y_i|} < \epsilon$$

where ϵ is some predefined constant. If you want the square root to be accurate to five decimal places then $\epsilon = 0.00001$. Define a word named `SQROOT` that calculates the floating-point square root of the floating-point number on top of the stack, using the iterative technique described above. Be sure to handle negative arguments! For example

```
81.0  SQROOT  F.      should print    9.0
2.0   SQROOT  F.      should print    1.41421
```

- 13) Define a word named `ECALC` that calculates e (the base of the natural logarithms, 2.718281828459045) using the infinite series

$$e = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \frac{1}{1 \cdot 2 \cdot 3 \cdot 4} + \dots$$

There should be no storing of intermediate results in memory locations (use the stack!). This is another example of an iterative algorithm.

- 14) Using the formula

$$n! = \text{factorial}(n) = 1 * 2 * 3 * \dots * n$$

define a word named `FACT` that calculates and prints the single-word integer factorial of the single-word integer value(n) on top of the stack. If n is not in the range

$$0 \leq n \leq 7$$

output an error message instead of calculating the factorial.
(Note: $\text{factorial}(0) = 1$ by definition).

15) Why is 7 used as the upper limit of n in the previous example?

16) As you can see from the previous examples the value of factorial(n) grows very rapidly as n increases. If we are interested only in an approximation to factorial(n), instead of its exact integer value, we can use Stirling's approximation to n!

$$\sqrt{2n\pi} \left(\frac{n}{e}\right)^n < n! < \sqrt{2n\pi} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n-1}\right)$$

(where $e = 2.71828182$ and $\pi = 3.14159265$). Define a word named N! which calculates the upper and lower bounds of n! where n is the single-word integer on top of the stack. Use the E format for printing the values. Note that the word x**y from exercise 6 above must also be used here. For example

3	N!	should print	5.8362 E 0	<	3!	<	6.0030 E 0
4	N!	should print	2.3506 E 1	<	4!	<	2.4006 E 1
5	N!	should print	1.1802 E 2	<	5!	<	1.2002 E 2
100	N!	should print	9.3248 E 157	<	100!		
						<	9.3326 E 157

17) Define a word named STAT which inputs a series of numbers and calculates their mean and standard deviation. When the word is executed it should first ask the operator how many numbers are to be input (a single-word integer) and then input each (floating-point) number. After the last number has been entered the mean and standard deviation should be calculated using the formulas below, and then output.

$$\text{mean} = \frac{\sum X_i}{n}$$

$$\text{std. dev.} = \sqrt{\frac{n\sum(X_i)^2 - (\sum X_i)^2}{n(n-1)}}$$

where n is the number of items input and X_i is each separate number that is input. For example

```

STAT  HOW MANY NUMBERS?  4
?4.0
?6.0
?4.0
?6.0
MEAN  =   5.0
STD DEV  =   1.1547

```

- 18) The greatest common division (GCD) of two integers a & b is the largest integer that evenly divides both a and b . For example, $\text{GCD}(20,25) = 5$; $\text{GCD}(20,40) = 20$. A classical algorithm found in computer science texts is Euclid's algorithm for finding the GCD. This algorithm may be stated as:

```

└─▶  $r = (a \bmod b)$       (i.e. -  $r$  equals the remainder
                        of  $a$  divided by  $b$ )
    If ( $r=0$ ) then the answer is  $b$ , stop.
     $a = b$ 
     $b = r$ 
    └─loop around

```

For example, we find that $\text{GCD}(27,21) = 3$ as follows

$a =$	27	21	6	
$b =$	21	6	3	← answer
$r =$	6	3	0	

(Note that this algorithm works regardless whether $a > b$ or $b > a$ initially.)

Write a word in FORTH named `GCD` that expects two single-word integers on the stack and then calculates and prints their greatest common divisor. For example, entering `27 21 GCD` should print `3`. Calculate `GCD(2166, 6099)`.

- 19) Frequently one encounters the problem of calculating the mean of a large group of single-word integers. These values could, for example, be a sequence of integer data points read in by the program from some data collection instrument. Assume that we have a vector defined as

```
2047 ( )DIM DATA
```

containing 2048 points. Define a word named `M+RMS` that calculates and prints both the mean and the rms (root mean squared) of the data vector. The rms is defined as the square root of the average of the squares of each point. One would like to avoid the use of floating-point arithmetic as on a mini-computer without floating-point hardware these operations take considerable time to execute. This additional time is even more noticeable when a large vector of numbers is being operated on. However, single-word integer arithmetic is definitely unacceptable for both the mean and the rms as there is an almost definite probability that the sum of a couple of numbers will overflow a single-word value. If we think of using double-word integer arithmetic then consider the square of the largest data point (32,767)--its value is 1,073,676,289 which is almost the largest value that may be stored in a double-word integer (Section 7.2). We then see that forming the sum of the squares of a vector of integers could possibly overflow a double-word sum. We could probably form the sum of all the numbers (for the average) as a double-word sum as it would take 32,768 numbers, each having the largest possible value (32,767) to overflow a double-word sum. To do this and perform

only one pass over the data would require that we keep both a floating-point sum and a double-word integer sum on the stack together. Manipulating these two sums in an alternate manner is very clumsy in FORTH so we resign ourselves to using only floating-point arithmetic.

In order to make this word as general as possible define the word

```
0 CONSTANT NOP
```

to contain the number of points (1-2048) in the vector DATA. Set NOP to 5 and calculate both the mean and rms of the data points

```
16,105
18,291
14,333
17,015
15,280
```

- 20) The algorithm used in the previous problem to calculate the mean of a group of integers (forming the sum of all the numbers and then dividing by the number of points) is not always satisfactory. One can conceivably imagine a very large set of data points (say on the order of one million points) in which case neither a double-word sum nor a floating-point sum would suffice. The double-word sum would overflow and the floating-point sum would get so large that the data points being added to it would be equivalent to zero (a floating-point number has only a finite precision). One solution to this problem is the stable running mean algorithm. If we have a sequence of data points X_i we calculate the new mean of the sequence X_1, X_2, \dots as

$$\text{newmean} = \text{oldmean} + \left(\frac{X_i - \text{oldmean}}{i} \right)$$

Initially we must set oldmean = 0. For example if we use the data points given in the previous problem we obtain

$$\begin{aligned}\text{newmean} &= 0 + \left(\frac{16105 - 0}{1} \right) = 16105 \\ \text{newmean} &= 16105 + \left(\frac{18291 - 16105}{2} \right) = 17198 \\ \text{newmean} &= 17198 + \left(\frac{14333 - 17198}{3} \right) = 16243\end{aligned}$$

and so on. If we calculate the average of these three data points it also equals 16243, as we would expect. This algorithm is termed a "running mean" since we maintain the current mean at each iteration. Here we do not have to worry about overflow as long as our data points are single-word integers. (One problem that may be encountered here is a sequential loss of precision in the integer divisions. This effect may be lessened by use of floating-point arithmetic.) Refine a word named RMEAN that calculates the mean of NOP number of points in the vector named DATA (see previous problem). Test this word using the five data points from the previous problem. Keep the values of newmean and oldmean on the stack!

- 21) The following algorithm calculates the date of Easter for any year after 1582 [Knuth, Donald E., The Art of Computer Programming, Vol. 1, Addison-Wesley, 1973]. Define a word named EASTER which prints the date of Easter, given the integer year on top of the stack. For example

1977 EASTER should print APRIL 10, 1977
1978 EASTER should print MARCH 26, 1978

(All division in the algorithm is integer division).

$G = (\text{YEAR} \bmod 19) + 1$ (G is the golden number of the year
in the 19- year Metonic cycle)
 $C = (\text{YEAR} / 100) + 1$ (C is the century)
 $X = ((3 * C) / 4) - 12$ (correction for number of years in
which leap year was dropped)
 $Z = (((8 * C) + 5) / 25) - 5$ (correction to synchronize Easter
with the moon's orbit)
 $D = ((5 * \text{YEAR}) / 4) - X - 10$ (find Sunday)
 $E = ((11 * G) + 20 + Z - X) \bmod 30$
 IF $(E < 0)$ $E = E + 30$
 IF $((E = 25) \text{ AND } (G > 11)) \text{ OR } (E = 24)$ $E = E + 1$
 (E is the epact which specifies
when a full moon occurs)
 $N = 44 - E$
 IF $(N < 21)$ $N = N + 30$ (The N^{th} of March is a full moon)
 $N = N + 7 - ((D + N) \bmod 7)$ (Advance to Sunday)

 IF $(N > 31)$ then the date is April $(N - 31)$
 else the date is March N

Define the necessary variables to be used as intermediate results
 and also try to use the stack as efficiently as possible.

16. REAL-TIME I/O

This chapter is the essence of what FORTH was originally designed for - the control of real-time data acquisition devices. The definition of the word "real-time" is itself quite obscure in the development of computing technology and is a frequently misused adjective. In this author's opinion a real-time device is one which, when it presents data to the computer, must be acknowledged by the computer within a certain limited time frame or else the original data is lost. This loss of data generally occurs because the device has already accumulated new data for the program. The speeds of real-time devices vary considerably and it is the speed of the computer along with the speed of the real time device(s) that determines the maximum data acquisition rate of a particular system.

16.1 INTERRUPTS

An *interrupt* is the facility whereby a device notifies the computer that the device requires service of some sort. Typical reasons for a device generating an interrupt are:

- the device has some data for the computer to input,
- the device has completed the output of the previous data and is ready to accept some additional data to output,
- the device has detected an error of some sort during an I/O operation.

Once the computer is notified that a device requires service, the computer can start a program to handle the device.

From a programming standpoint it is one's job to write a program that will service a specific device's interrupt. In reality the program that services a device's interrupt is generally only a small portion of a larger program and is therefore called an *interrupt routine*.

The nitty gritty details of exactly how a device generates an interrupt followed by how the computer determines exactly which device is requesting the service are vastly different for each computer and beyond the scope of this primer. Suffice it to say that with KPNO Varian FORTH one writes a word to process a specific device's interrupt and FORTH will then perform the required linkage to insure that this word will get control every time the specified device generates an interrupt. The actual definition of this word will be discussed in greater detail in the next section.

A CAMAC interrupt (CAMAC is described in the next section) is referred to as a LAM which means that the device is telling the computer "Look At Me".

16.2 CAMAC I/O

One problem that has plagued the users of mini-computers ever since their introduction has been the interfacing (i.e. - the electronic connection) of devices to a specific computer. Typical problems in this area are

- (a) a particular type of device requires a different interface for every different computer,
- (b) when you switch computers you must buy all new interfaces for your devices (a favorite technique of mini-computer manufacturers to "lock" a customer into their series of computers).

As a way around these problems the CAMAC standards were developed by the European Standards on Nuclear Electronics Committee (ESONE) in 1969. These standards have since been updated and recently adopted by the IEEE.

CAMAC is a hardware system designed to provide simple, computer-independent input and output. Standardized electronic components (modules) are joined together in one or more machine-independent

housings (crates) which are then connected to a specific computer via a single machine-dependent interface (the Branch Driver). The advantage in using CAMAC is that over 100 different devices may be connected to the computer through a single interface, thus simplifying the possible transition to a new computer. Additionally, there are over 70 companies worldwide producing CAMAC hardware thereby relieving the user from having to design and build his own specialized devices.

Unfortunately in discussing any type of input/output, one must become more familiar with the specifics of the computer being utilized. In the case of this primer it requires that one understand that the computers in use at KPNO are Varian 620's with a word size of 16-bits. Thus the previous references to a single-word integer are references to a 16-bit integer and the double-word integer is really a 32-bit integer. All data transfers between the Varian and CAMAC involve 24-bits.

The CAMAC Module is the device that the program wishes to control - it is the module that we must initialize when an I/O operation is to commence and it will be the module that will generate the interrupt when the I/O operation is finished. At a lower level, each module may be sub-addressed, to facilitate the control of multi-channel modules. As an example, the KPNO Timer II is a dual channel, high resolution timer and both channels are completely independent of each other (but both channels are packaged in a single module). To specify which channel you wish to control you must specify both the module and the sub-address; likewise when processing an interrupt from the timer you must determine which channel (i.e., sub-address) generated the interrupt. When addressing a CAMAC module a sub-address value between 0-15 (decimal) will be specified. In the programming of modules which do not require a sub-address, zero is commonly used. It should be noted that the sub-address does not always address separate channels in a multi-channel

module, instead in some modules the sub-address specifies additional functions for the module to perform (the KPN0 Input/Output Register is a good example).

Modules are housed together in CAMAC Crates with up to 23 modules per crate. Each module in a specific crate is addressed by specifying the slot number in the crate of the module. These slot numbers are also referred to as station numbers and have a value between 1-23 (decimal). Some modules physically require more than one slot in a crate (due to the width of the module) and the KPN0 Display Panel Controller Module is an example of a module requiring two slots. These multi-slot modules are addressed by specifying only the lowest slot number of the module (for example the Display Panel usually occupies slots 20 and 21 and is addressed through slot 20).

There may be up to seven crates in a CAMAC system and these crates are addressed as 1-7.

The addressing required to select a specific module is therefore:

$$\begin{aligned}C &= \text{Crate (1 - 7)} \\N &= \text{Station Number (1 - 23)} \\A &= \text{Sub-Address (0 - 15)} \\F &= \text{Function Code (0 - 31)}\end{aligned}$$

This sequence is generally referred to as CNAF.

The Function Code is the method whereby you tell the selected module exactly what function it is to perform. There may be up to 32 (decimal) different Function Codes for a module (specified as 0-31) and although the codes will differ from one module to another, the general convention that is followed is:

Function	0 - 7	→	read
	8 - 15	→	control
	16 - 23	→	write
	24 - 31	→	control

16.3 FORTH CAMAC WORDS

The programming of CAMAC input and output is actually much simpler than would appear from the previous section since FORTH handles most of the complicated details. The programming of a module from a colon definition is the subject of this section.

Each specific CAMAC module is identified in FORTH as a word which identifies that specific module. In order to define a <module-ID> one writes

```
<slot-number> $CN; <module-ID>
```

where <slot-number> is a single-word integer value between 1 and 23 specifying the slot number in the crate. <module-ID> is the name assigned by the programmer to the module. For example, the following are from block 54:

```
13 $CN; $1IO      ( I/O REGISTER 1 )
14 $CN; $2IO      ( I/O REGISTER 2 )
18 $CN; $TM       ( TIMER MARK II )
19 $CN; $DO       ( DIGITAL OSCILLATOR )
```

One should note that the convention used at KPN0 is to prefix all CAMAC words with a dollar sign. Before executing the above definition one must store in the integer CRATE the crate number of the <module-ID> being defined. The definition of CRATE is

```
1 VARIABLE CRATE
```

and this word is defined in block 50 (and will therefore be entered into the dictionary when the CAMAC blocks, 50 thru 53, are loaded when the word USER is executed).

Consider the example definition

```
5  CRATE  !  
19  $CN;   $5DO
```

which defines the word `$5DO` as the device in slot 19 of crate 5.

Now that we know how to identify each module we need a way to have a specific module execute a specific function. As would be expected, one defines a word in FORTH which when executed will perform the specified CAMAC I/O function. The general format of the definition is

<F-code> <sub-address> <module-ID> <operation-type> <name>

<F-code> Is a single-word integer value (between 0 and 31) that specifies the function code.

<sub-address> Is a single-word integer value (between 0 and 15) that specifies the sub-address.

<module-ID> Is a previously defined word which specifies both the crate and the station number of the module (described above).

<operation-type> Is a system defined word (refer to the list below) which specifies what parameters are expected to be on the stack (prior to an output operation) or what parameters will be left on the stack (following an input operation). This word processes all the previous parameters (<F-code>, etc.) and creates the appropriate dictionary entry for <name>.

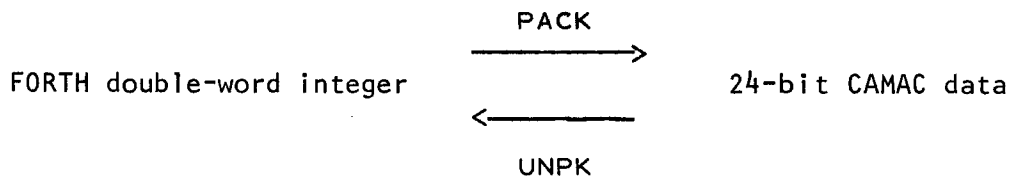
<name> Is the user specified identifier which will identify this specific I/O operation. When the word <name> is executed the I/O operation will be performed.

The following words are defined as <operation-types>:

\$COMMAND;	Transmits a command to the module - no stack operations performed.
\$ACOMMAND;	Expects a single-word integer sub-address on top of the stack, then transmits a command to the module.
\$READ;	Reads the low-order 16-bits of CAMAC data onto the top word of the stack.
\$AREAD;	Expects a single-word integer sub-address on top of the stack, then reads the low-order 16-bits of CAMAC data onto the top word of the stack.
\$2READ;	Reads the full 24-bits of CAMAC data onto the top two words of the stack (see below for format).
\$2AREAD;	Expects a single-word integer sub-address on top of the stack, then reads the full 24-bits of CAMAC data onto the top two words of the stack (see below for format).
\$WRITE;	Writes the word on top of the stack as the low-order 16-bits of CAMAC data.
\$AWRITE;	Expects a single-word integer sub-address on top of the stack, then writes the next word on the stack as the low-order 16-bits of CAMAC data.
\$2WRITE;	Write the two words on top of the stack as the full 24-bits of CAMAC data (see below for format).
\$2AWRITE;	Expects a single-word integer sub-address on top of the stack, then writes the next two words on the stack as the full 24-bits of CAMAC data (see below for format).

Note that the A-command format for each <operation-type> allows the specification of the sub-address when the word is *executed*, whereas the other format requires the sub-address to be specified when the word is *defined*. When defining a word with the A-command format the <sub-address> in the definition *must* be specified as zero.

Unfortunately, due to a peculiarity of the Varian hardware in processing double-word integers, the format of the 24-bit CAMAC data does not correspond to the format of FORTH's double-word integers. In order to convert between the two formats, the words `PACK` and `UNPK` are provided:



Whenever 16-bits of CAMAC data are transferred a single-word integer is used.

The following examples of some CAMAC definitions assume that the reader has available the write-ups on the specific module (which describes the interpretation of the <F-code> and <sub-address> for the specific module in question).

```
8 0 $TM $ACOMMAND; +MOVE
```

Defines the word `+MOVE` for the timer module as `F(8)` (i.e. - the <F-code> 8). This F-code will set the sign line to plus. Note that this word, when executed, expects the top of the stack to specify the sub-address, hence this word may be used for either channel of the timer.

1 +MOVE Sets the sign of the timer Channel 1 to plus.

2 +MOVE Sets the sign of the timer Channel 2 to plus.

3 +MOVE Sets the sign of both Channel 1 and Channel 2 of
the timer to plus.

16 15 \$UD \$2WRITE; UDCWRITE
Defines the word UDCWRITE for the up/down counter
as F(16). This F-code writes 24-bits of data to the
up/down counter. Note that the sub-address is
specified as 15 (which equals $8 + 4 + 2 + 1$) therefore
this command will write to channels 1, 2, 3 and 4.

50000 , PACK UDCWRITE
Writes 50,000 to channels 1, 2, 3 and 4 of the
up/down counter. Note that the double-word integer
must be PACKed prior to the I/O operation in
order to convert it to the 24-bit CAMAC data format.

2 0 \$UD \$2AREAD; UDCREAD
Defines the word UDCREAD for the up/down counter
as F(2). This F-code reads 24-bits of data from the
specified channel(s) of the up/down counter onto the
stack.

8 UDCREAD UNPK D.
Reads the contents of the up/down counter's channel 4
counter onto the stack, converts the 24-bit CAMAC
word into a double-word integer and then prints
the result.

16 2 \$DO \$WRITE; DDOK2
Defines the word DDOK2 for the DD0 module as F(16).
This F-code writes 16-bits of data to channel 2 of
the DD0 as the K-factor.

Writes 256 as the K-factor for channel 2 of the DD0.
 Note that no conversion is required when reading or
 writing a single-word integer as 16-bits of CAMAC data.

16.4 FORTH INTERRUPT WORDS

The method of writing a sequence of FORTH words to process interrupts from a specific device is to use the words `!:` and `;!C` to begin and terminate the definition, as follows:

$$\text{<module-ID> } \$! \quad !: \quad \text{<name> } \text{<words> } \left\{ \begin{array}{l} ;! \\ ;!C \end{array} \right\}$$

`<module-ID> $!`

Identifies the device whose interrupts are to be processed. (`<module-ID>` was described in Section 16.2).

`!:`

Starts the interrupt colon definition, similar to the standard `:`

`<name>`

Is the user specified identifier that identifies the dictionary entry for this definition.

`<words>`

Are the names of previously defined FORTH words that will be executed when an interrupt occurs from the specified device.

`;!:`

Terminates the interrupt colon definition, similar to the standard `;`

`;!C`

Terminates the interrupt colon definition, identical with `;!:` however, `;!C` will cause a Branch Driver stack "pop" to be executed before returning to the interrupted routine. (This is the normal way to terminate an interrupt routine as the Branch Driver stack is automatically "pushed" when an interrupt is acknowledged.)

Once this definition is entered into the dictionary, all interrupts from the specified device will cause the specified sequence of <words> to be executed.

The following example should elucidate many of the techniques in this chapter. We want to write a word that keeps the time-of-day and will print out the current time-of-day on demand. In order to "count" the time we will use the KPN0 timer module to generate an interrupt every hundredth of a second (0.01 second). This interrupt routine will increment a double-word integer once every 0.01 seconds and the word WHATTIME will print the contents of this counter when executed. Additionally we need a word SETTIME which initializes the counter to a specified time.

First the CAMAC function words:

```

0   1   $TM   $READ;      TMRECLAM
11  1   $TM   $COMMAND;   TMGO
16  1   $TM   $2WRITE;    TMLOADPERIOD
17  1   $TM   $2WRITE;    TMLOADN
25  1   $TM   $COMMAND;   TMCLEAR
27  1   $TM   $COMMAND;   TMENABLELAM

```

The double-word integer counter:

```

DPREC
0, 2VARIABLE COUNTER

```

The interrupt processing routine, named TODINT:

```

$TM $! !: TODINT TMRECLAM DROP COUNTER D@
      1 M+ COUNTER D! ;!C

```

The word to output the current time-of-day:

```
: WHATTIME 2 F ! ( field width of 2 for numeric output )  
  
    COUNTER D@ 1 100 D*/  
        ( convert 0.01 sec. to sec. )  
  
    60 M/MOD 60 /MOD ( hours, min., sec. )  
  
    S. " ;" S. " ;" S. ;
```

The word to initialize the timer and input the current time-of-day from the operator:

```
: SETTIME ." ENTER TIME (HH:MM:SS.) "  
  
    DASK 100 1 D*/ ( convert sec. to 0.01 sec. )  
  
    COUNTER D! ( initialize counter )  
  
    10000 , PACK TMLoadPERIOD  
        ( 10,000 micro-sec. = 0.01 sec. )  
  
    999999 , PACK TMLoadN ( run for a long time )  
  
    TMENABLELAM ( one interrupt on every pulse )  
  
    TMGO ; ( start the timer )
```

Before executing these words one must execute

```
$SETUP
```

which initializes the CAMAC system.

The following points should be noted in the example:

- Channel 1 of the timer is arbitrarily used (channel 2 could just as easily have been used).
- The double-word integer COUNTER is effectively counting the number of 0.01 seconds past midnight.
- The interrupt routine simply clears each LAM and increments the double-word counter.
- Since there are no double-word integer multiply and divide words (corresponding to * and F*, / and F/) the word D*/ is used to multiply and divide double-word integers.
- The word SETTIME initializes the timer rate to 0.01 seconds (note the output to the timer must be specified in micro-seconds = 10^{-6} seconds) and sets the number of interrupts to 999,999. This latter number has no particular meaning in this example except to insure a sufficient number of interrupts (999,999 interrupts divided by 100 interrupts per seconds = 9999 seconds \cong 3 hours).

The reader should enter these words into a block and then execute them to confirm that they perform as claimed. Furthermore you should thoroughly understand the example since many various techniques are used.

"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to," said the Cat.

"I don't much care where--" said Alice.

"Then it doesn't matter which way you go," said the Cat.

"--so long as I get *somewhere*," Alice added as an explanation.

"Oh, you're sure to do that," said the Cat, "if you only walk long enough."

LEWIS CARROLL

Alice's Adventures in Wonderland

APPENDIX A - ASCII CHARACTER SET

7-Bit Octal	Character	8-Bit Octal with Parity		7-Bit Octal	Character	8-Bit Octal with Parity	
		Even	Odd			Even	Odd
000	NUL (null)	Control/Shift-P	000 200	100	@ (at sign)	300 100	
001	SOH (start of header)	Control-A	201 001	101	A (upper case alphabets)	101 301	
002	STX (start of text)	Control-B	202 002	102	B	102 302	
003	ETX (end of text)	Control-C	003 203	103	C	303 103	
004	EOT (end of transmission)	Control-D	204 004	104	D	104 304	
005	ENQ (enquiry)	Control-E	005 205	105	E	305 105	
006	ACK (acknowledge)	Control-F	006 206	106	F	306 106	
007	BEL (ring bell)	Control-G	207 007	107	G	107 307	
010	BS (backspace)	Control-H	210 010	110	H	110 310	
011	HT (horizontal tab)	Control-I	011 211	111	I	311 111	
012	LF (line feed)	Control-J	012 212	112	J	312 112	
013	VT (vertical tab)	Control-K	213 013	113	K	113 313	
014	FF (form feed, top of page)	Control-L	014 214	114	L	314 114	
015	CR (carriage return)	Control-M	215 015	115	M	115 315	
016	SO (shift out)	Control-N	216 016	116	N	116 316	
017	SI (shift in)	Control-O	017 217	117	O	317 117	
020	DLE (data link escape)	Control-P	220 020	120	P	120 320	
021	DC1 (device control 1)	Control-Q	021 221	121	Q	321 121	
022	DC2 (device control 2)	Control-R	022 222	122	R	322 122	
023	DC3 (device control 3)	Control-S	223 023	123	S	123 323	
024	DC4 (device control 4)	Control-T	024 224	124	T	324 124	
025	NAK (negative acknowledgment)	Control-U	225 025	125	U	125 325	
026	SYN (synchronize)	Control-V	226 026	126	V	326 126	
027	ETB (end of transmission blk)	Control-W	027 227	127	W	327 127	
030	CAN (cancel)	Control-X	030 230	130	X	330 130	
031	EM (end of medium)	Control-Y	231 031	131	Y	131 331	
032	SUB (substitute)	Control-Z	232 032	132	Z	332 132	
033	ESC (escape)	Control/Shift-K	033 233	133	[(left bracket)	333 133	
034	FS (file separator)	Control/Shift-L	234 034	134	\ (back slash)	134 334	
035	GS (group separator)	Control/Shift-M	035 235	135] (right bracket)	335 135	
036	RS (record separator)	Control/Shift-N	036 236	136	^ (up arrow)	336 136	
037	US (unit separator)	Control/Shift-O	237 037	137	_ (back arrow)	137 337	
040	(space)		240 040	140	^ (accent grave)	140 340	
041	! (exclamation point)		041 241	141	a (lower case alphabets)	341 141	
042	" (quote)		042 242	142	b	342 142	
043	# (pound sign)		243 043	143	c	143 343	
044	\$ (dollar sign)		044 244	144	d	344 144	
045	% (percent sign)		245 045	145	e	145 345	
046	& (ampersand)		246 046	146	f	346 146	
047	' (prime)		047 247	147	g	347 147	
050	((left paren)		050 250	150	h	350 150	
051) (right paren)		251 051	151	i	151 351	
052	* (asterisk)		252 052	152	j	352 152	
053	+ (plus sign)		053 253	153	k	153 353	
054	, (comma)		254 054	154	l	354 154	
055	- (minus sign, hyphen)		055 255	155	m	355 155	
056	. (period)		256 056	156	n	356 156	
057	/ (slash)		057 257	157	o	357 157	
060	0 (numerics)		060 260	160	p	360 160	
061	1		261 061	161	q	161 361	
062	2		262 062	162	r	362 162	
063	3		063 263	163	s	163 363	
064	4		264 064	164	t	364 164	
065	5		065 265	165	u	365 165	
066	6		266 066	166	v	366 166	
067	7		067 267	167	w	367 167	
070	8		270 070	170	x	370 170	
071	9		071 271	171	y	371 171	
072	: (colon)		272 072	172	z	372 172	
073	; (semi-colon)		273 073	173	{ (left brace)	373 173	
074	< (less than)		074 274	174	(vertical bar/logical OR)	374 174	
075	= (equals sign)		275 075	175	} (right brace)	375 175	
076	> (greater than)		276 076	176	~ (tilde)	376 176	
077	? (question mark)		077 277	177	DEL (delete, rub out)	377 177	

APPENDIX B - FORTH ERROR CODES

Whenever FORTH detects an error, a message is output to the terminal consisting of a question mark followed by a single character. This appendix describes the error associated with each single character code.

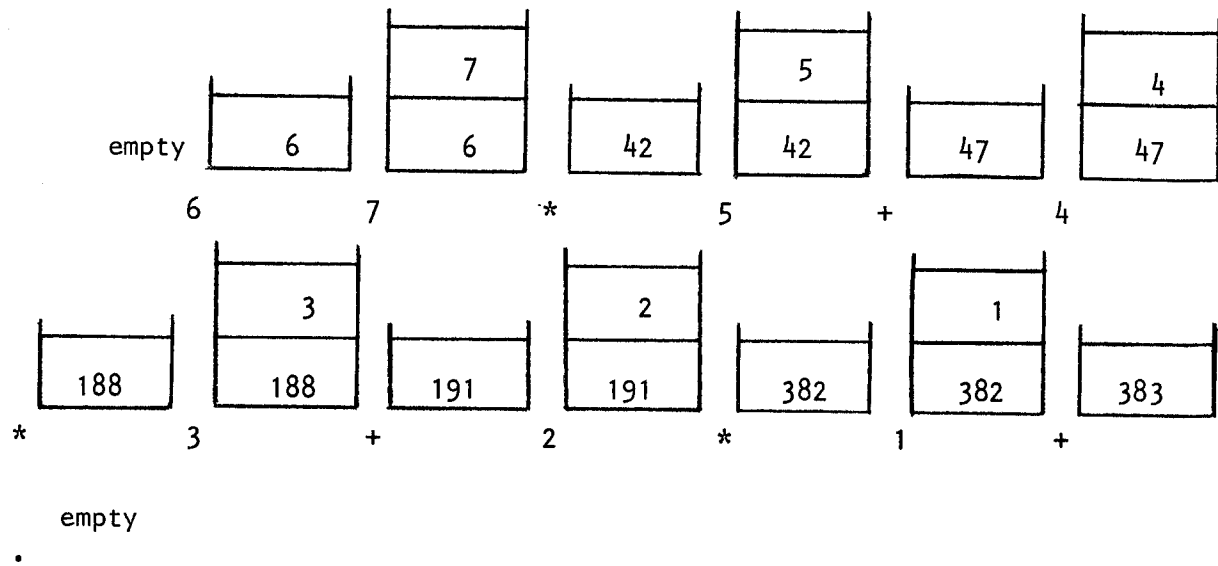
- ?Q The word could not be found in the dictionary. Check for a possible typing error or a spelling error.
- ?R Line printer off-line or disc error. In the case of a disc error, the sequence `N 6 + @ D` will print the disc status word.
- ?S Program abort (the word `ABORT` was executed by someone to exit some piece of code).
- ?T Magnetic tape error detected by the FORTH Block I/O drivers (probably the tape drive is off-line or there is no write-ring on a write operation).
- ?U Stack underflow. A word or words were expected on the stack but the stack was empty.
- ?V Dictionary and stack overflow. The combined size of the dictionary and the stack exceeds the total allocated core area. This error indicates either too many words have been entered into the dictionary or else someone is pushing too many words onto the stack.
- ?W Illegal disc block access. The block number of the requested disc block or tape block is illegal (i.e., - not in the range 0-4895).
- ?X Input error following an asking-word request for a number or an unexpected interrupt from some device.
- ?Y A magnetic tape error of some sort (tape drive off line, parity error, missing write ring on write, etc.) has been detected by the direct magnetic tape drivers.
- ?Z Indicates a console interrupt (operator pressed Control-X key), a display panel interrupt (operator pushed the rightmost bottom pushbutton) or power-fail recovery.

APPENDIX C

ANSWERS TO EXERCISES

CHAPTER 5

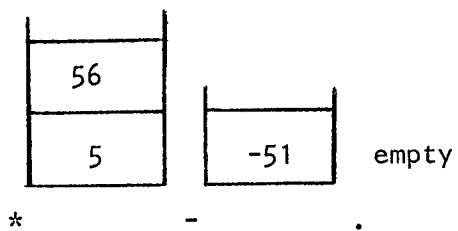
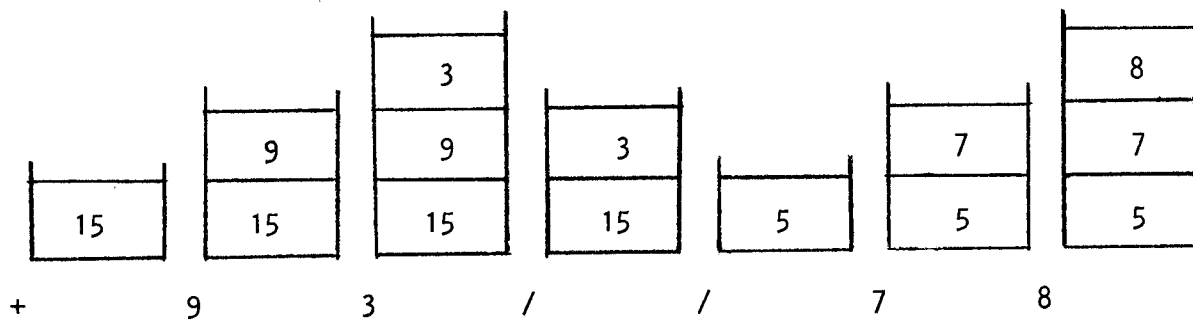
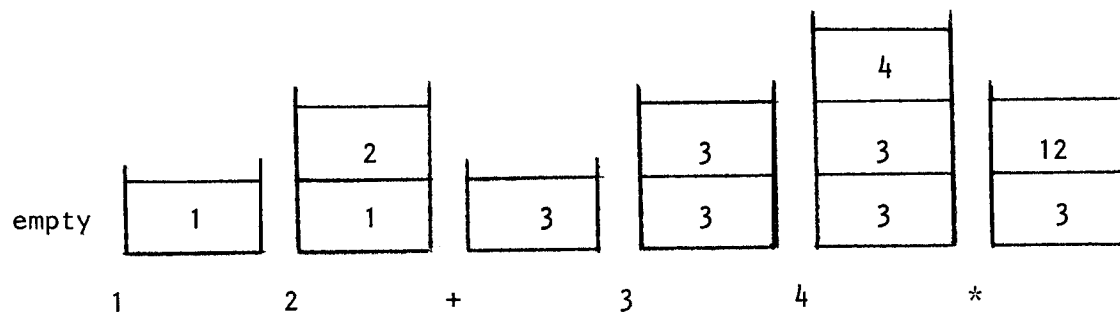
1) 6 7 * 5 + 4 * 3 + 2 * 1 + . yields 383



2) 1 2 + 3 4 * + 9 3 / / 7 8 * - . yields -51

CHAPTER 5

2) Continued



CHAPTER 6

- 1) a) SINY
 b) X+Y
 c) XANT
 d) XANT
 e) SINY
 f) none, will generate Y+XX ?Q

CHAPTER 7

```
1)      5 VARIABLE   I
        100 CONSTANT J

        I @ 1 + I !
        I @ 5 * ' J !
        I @ .
        J .
        J J * 10 - I !
        I @ J / 2 + ' J !
        I @ .
        J .
```

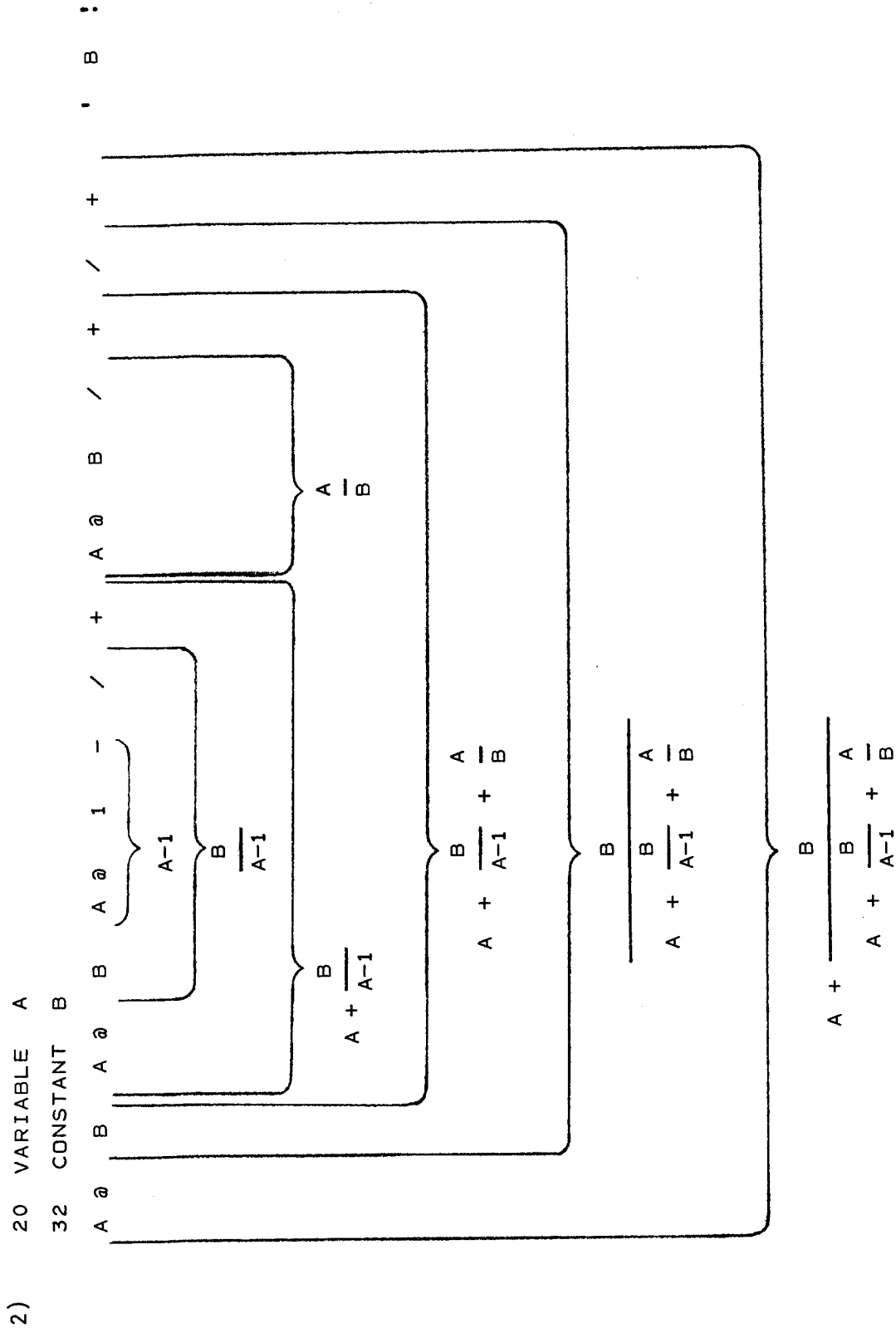
will print 6

will print 30

will print 890

will print 31

CHAPTER 7



B . will print the value 21

CHAPTER 7

3)

```
10 , 2VARIABLE  II
30 , 2 CONSTANT  JJ
JJ  1,  D+  II D!
JJ  II D@ D-  ' JJ D!
II D@ D. will print 31.
JJ D. will print -1.
II D@ JJ D+ 101, D+ II D!
JJ II D@ D+ ' JJ D!
II D@ D. will print 131.
JJ D. will print 130.
```

4)

```
1  CONSTANT  A
5  VARIABLE  B
8  CONSTANT  C
25 VARIABLE  D

B @ C D @ + * ' A !
A . will print 165

B @ C * B @ D @ * + ' A !
A . will print 165
```


CHAPTER 7

- 5) 3 , 2CONSTANT A
 5 , 2VARIABLE B
 7 , 2CONSTANT C
 11 , 2VARIABLE D
 17 , 2CONSTANT E

B D@ C D+ D D@ D+ E D+ ' A D!

A D.

will print 40.

B D@ C D D@ E D+ D+ D+ ' A D!

A D.

will print 40.

- 6) $(B^2 - 4AC)$ whose value will be 36.0. This value is left on the stack (since it is not specifically stored in a variable).
- 7) For a single-word integer the largest value is 32,767 which corresponds to 9:06:07. For a double-word integer the largest value is 1,073,741,823 which corresponds to approximately 298,261 hours! Hence if you are counting the number of seconds past midnight you must use a double-word integer since a single-word integer does not provide sufficient precision.

CHAPTER 7

8) I @ SFLOAT X F@ F* A D@ B D- DFLOAT F/
 J SFLOAT Y F/ F/ ' Y F!
 Y F. will print 0.12002

Note how the maximum accuracy (floating-point) is maintained throughout the calculation. There is no truncation performed until required (when the new value of B is stored).

- 9) a) false, since its value is 0.
 b) false, since (true false AND) ---> false.
 c) true, since (true true AND) ---> true.
 d) false, since (false false OR) ---> false.
 e) true, since (false true XOR) ---> true.
 f) true, since (true true OR) ---> true.

10) 2 since $10_{10} - 8_{10} = 2.$

CHAPTER 8

1) Use the formula $Z = (X - Y) + (X - Y)$.

X D@ Y D- X D@ Y D- D+ ' Z D!

Z D. will print -8.

X D@ Y D- 2DUP D+ ' Z D!

Z D. will print -8.

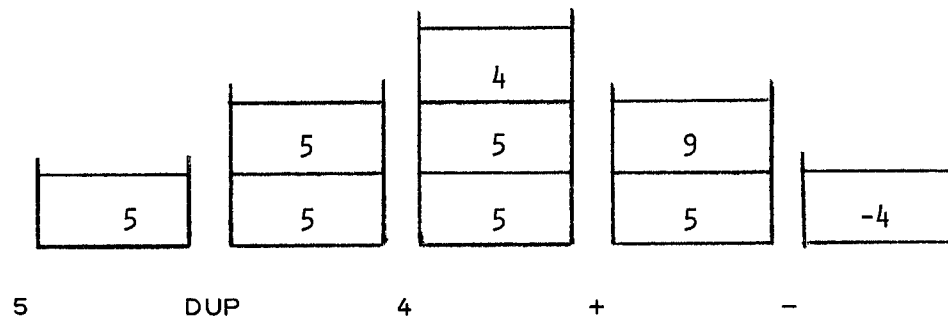
2) X D@ 2DUP Y D- 2SWAP Y D+ D+ ' Z D!

Z D. will print 10.

3) X D@ Y 2OVER 2OVER D- D+ D+ ' Z D!

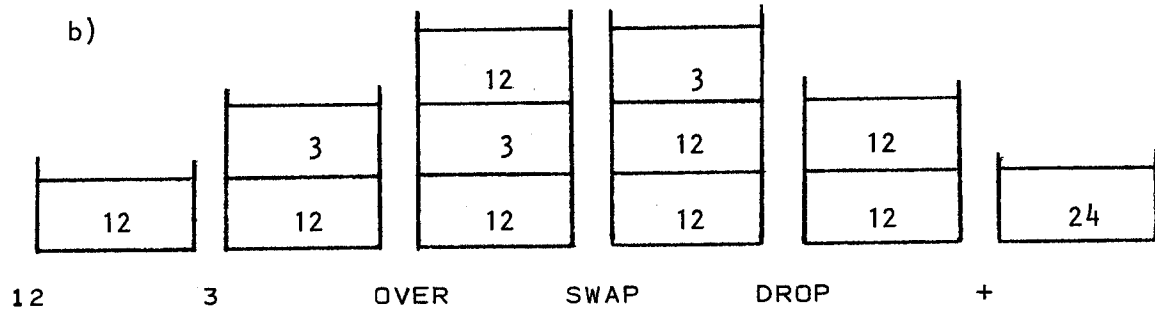
Z D. will print 10.

4) a)

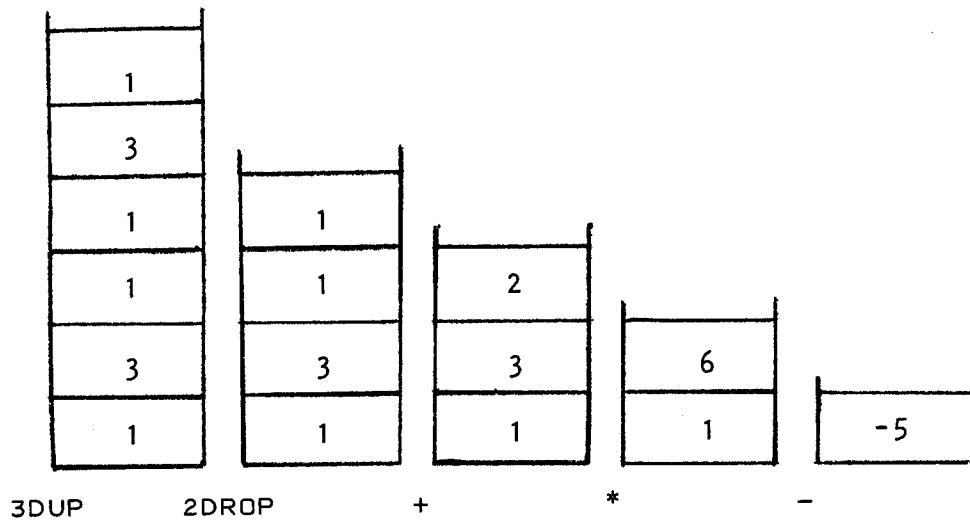
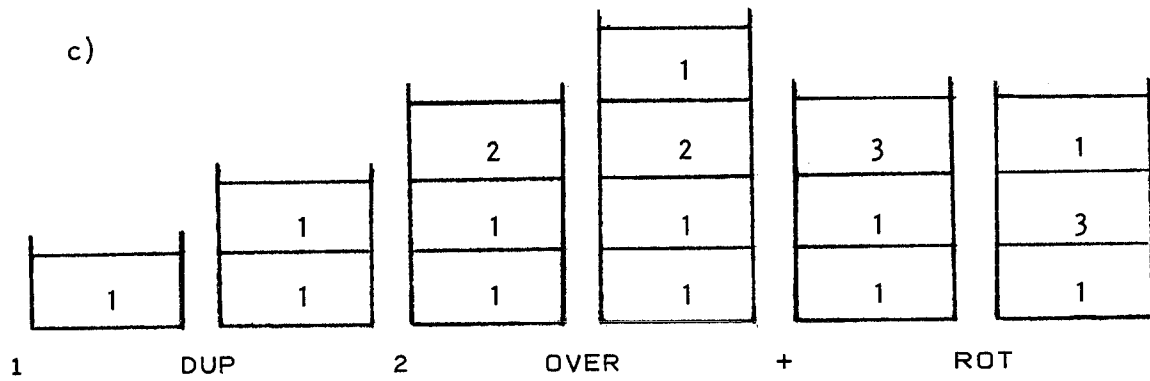


CHAPTER 8

4) b)

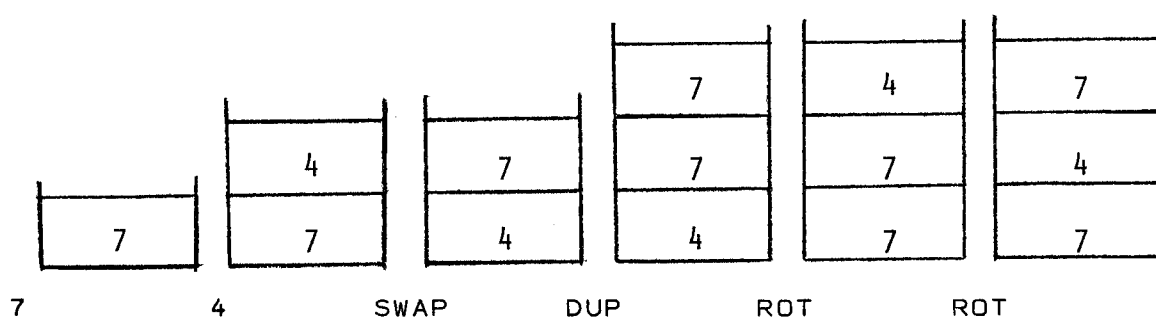
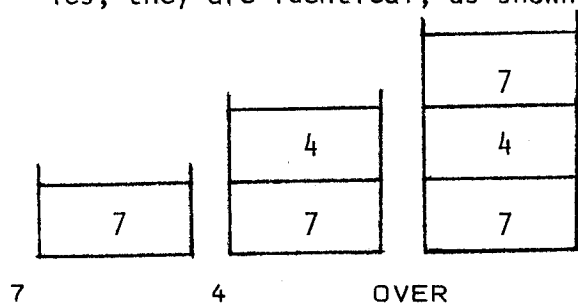


4) c)



CHAPTER 8

5) Yes, they are identical, as shown below:



6) : 2PICK DUP 1 + PICK SWAP PICK ;
 : 3PICK DUP DUP 2 + PICK SWAP
 1 + PICK ROT PICK ;

CHAPTER 9

1) : I**4 DUP DUP DUP * * * ;
 : I**4 DUP * DUP * ;

The second method is preferable since it requires one less multiplication than the first method.

CHAPTER 10

- 1) a) 2898, 2899
 b) -4, -5
 c) -2, -4, -6
 d) -3
 e) -3
 f) 18, 12, 6
 g) 6, 12
 h) -1

- 2) : 1PROD 1 .11 2 DO I * 2 +LOOP . ;

3840 will be printed.

Also the following word will work (why?)

: 2PROD 1 12 2 DO I * 2 +LOOP . ;

- 3) : 3PROD 1 2 10 DO I * -2 +LOOP . ;

3840 will be printed.

- 4) 0 VARIABLE INDEX

: INCINDEX INDEX @ 1 + INDEX ! ;

: SUM3 50 INDEX ! 0 BEGIN INDEX @ + INCINDEX INDEX
@ 100 > END . ;

- 5) : D0= OR 0= ;
 : D= D- D0= ;
 : D0< SWAP DROP 0< ;
 : D> 2SWAP D- D0< ;
 : D< D- D0< ;
 : DMAX 2OVER 2OVER D< IF 2SWAP THEN 2DROP ;
 : DMIN 2OVER 2OVER D> IF 2SWAP THEN 2DROP ;

CHAPTER 10

Note the order in which these words are defined, so that each word may use a previously defined word. The word `DO=` uses the fact that a double-word integer is zero if and only if both single-word halves of the double-word are zero. Thus the `OR` of these two halves of the double-word will be zero if and only if both halves are zero. The word `DO<` uses the fact that the sign of a double-word integer is contained in the top half of the double-word and therefore the bottom half may be ignored for this comparison.

- ```

6) : SIGN DUP 0= IF ." Z" DROP
 ELSE 0 < IF ." N"
 ELSE ." P"
 THEN
 THEN ;

7) : EX 1 + 1 DO I PICK . LOOP ;

8) 19 ()DIM VEC
 : VINIT 20 0 DO I I VEC ! LOOP ;

9) : (). 0 DO CR DUP @ . 1 + LOOP DROP ;
 : 2(). 0 DO CR DUP D@ D. 2 + LOOP DROP ;
 : 3(). 0 DO CR DUP F@ F. 3 + LOOP DROP ;

10) 5 3()DIM VC
 0 CONSTANT N

 29.7 0 VC F! -8.2 1 VC F! -1.9 2 VC F!
 4.5 3 VC F! 0.52 4 VC F! -8.3 5 VC F!

 : 3VBUBSORT ' N ! N 1 (1, 2, ... , N-1)
 DO I N 1 - (N-1, N-2, ... , 1)
 DO I 1 - VC F@ I VC F@ 3OVER 3OVER F>
 IF 3SWAP I VC F! I 1 - VC F!
 ELSE 3DROP 3DROP THEN
 -1 +LOOP
 LOOP ;

```



## CHAPTER 14

```
1) : IMIN SASK (PUSH FIRST NUMBER)
 BEGIN SASK DUP 0 <
 IF ." MIN VALVE =" SWAP . DROP 1
 ELSE MIN 0 THEN
 END ;
```

Note how both halves of the `IF` branch leave a number on the stack - 1 (if the terminating negative number is encountered) or 0 (if another number was compared). This 0 or 1 is then the <logical-condition> for the `END` word and only if the terminating negative number was entered does the `BEGIN - END` loop stop. This is a common programming practice in FORTH and the stack is a convenient place to put the "flag" value.

```
2) : FIB DUP 3 <
 IF ." N MUST BE GREATER THAN 3" DROP
 ELSE 0 1 (FIRST TWO FIB NUM) 0 .
 BEGIN DUP . (PRINT CURRENT)
 OVER OVER + ROT DROP
 (COMPUTE NEXT IN SEQ.)
 DUP 4 PICK >
 END 3DROP
 THEN ;
```

## CHAPTER 15

```
1) : ?ODD/EVEN 2 MOD IF ." ODD" ELSE ." EVEN" THEN ;

2) : ?TIME 60 M/MOD 60 /MOD . . . ;

3) 0.0 REAL A
 0.0 REAL B
 0.0 REAL C
 : DISCR B F@ 3DUP F* 4.0 A F@ F* C F@ F* F- ;
 : QUAD C F! B F! A F! CR DISCR 3DUP F0=
 IF 3DROP ." ONE REAL ROOT:"
 B F@ FMINUS 2.0 A F@ F* F/ F.
 ELSE 3DUP F0<
 IF ." TWO COMPLEX ROOTS "
 ." (REAL AND COMPLEX PARTS):"
 B F@ FMINUS 2.0 A F@ F* F/ F.
 FMINUS FSQRT 2.0 A F@ F* F/ F.
 ELSE ." TWO REAL ROOTS:" FSQRT 3DUP
 B F@ FMINUS 3SWAP F+ 2.0 A F@ F*
 F/ F.
 B F@ FMINUS 3SWAP F- 2.0 A F@ F*
 F/ F.
 THEN
 THEN
 THEN ;

4) : FDASINE 3DUP FABS 1.0 F=
 IF 90.0 F*
 ELSE 3DUP 3DUP 3DUP F* 1.0 3SWAP F-
 FSQRT FDATN
 3SWAP F0< IF 360.0 F- THEN
 THEN ;
```

## CHAPTER 15

```
5) : FDACQS 3DUP 3DUP F* 1.0 3SWAP F- FSQRT
 FDASINE 3SWAP F0<
 IF 180.0 3SWAP F- THEN ;
```

```
6) : X**Y 3OVER 0.0 F<
 IF ." NEGATIVE NUMBER ERROR" 3DROP 3DROP
 ELSE 3SWAP FLN F* FEXP THEN ;
```

Note that when using this algorithm  $2^{12} = 4095.99922$  and not 4096!  
This is due to the inexactness of both floating-point numbers and the exponential/logarithm functions. If one knew that the exponent were an integer then a combination of multiplies would generate the exact answer.

```
7) 1.0 3.0 F/ FCONSTANT 1/3
 : FCUBERT 3DUP 0.0 F<
 IF FMINUS 1/3 X**Y FMINUS
 ELSE 1/3 X**Y THEN ;
```

Note the technique used to obtain the maximum precision available for the infinite constant 0.333... By dividing 1.0 by 3.0 in the definition you obtain the maximum precision available, regardless which computer you are running on. If, however, you were to enter

```
0.33333 FCONSTANT 1/3
```

you would not be obtaining the maximum precision on any computer with more than 5 digits of precision. To a computer the numbers 0.3 and 0.33 are *not* equal!

## CHAPTER 15

8) : FROUND 0.5 F+ ;

9) : I\*\*J 2DUP SWAP SFLOAT FLN 4 PICK SFLOAT  
F\* FEXP 3SWAP 3DROP FROUND SFIX ;

Note the complications that arise in converting two single-word integers on the stack to floating-point, without storing either one in a temporary location. Of course, if this were a frequent operation one could define a new word to do it.

10) : CUBE DUP 0< IF MINUS 3 I\*\*J MINUS  
ELSE 3 I\*\*J THEN ;

11) The second definition is almost certainly preferable in all cases since it requires only two multiplications and no conversion back and forth to floating-point. Note also that one need not worry about the sign with the second definition. Finally, the first definition will take longer to execute and will be less precise since the exponential and logarithm functions require time to execute and neither of these are perfectly "exact" (integer multiply will always be exact).

12) 0.00001 FCONSTANT EPSILON  
: SQROOT 3DUP F0<  
IF "NEGATIVE ARGUMENT" 3DROP  
ELSE 3DUP 0.5903 F\* 0.4173 F+  
BEGIN 3OVER 3OVER F/ 3OVER F+ 0.5 F\*  
3SWAP 3OVER F- FABS EPSILON F<  
END 3SWAP 3DROP  
THEN ;

## CHAPTER 15

```
13) 0.000001 FCONSTANT EPSILON
 : ECALC 2.0 2.0 32767 3 DO 3OVER 1.0 3SWAP F/
 3OVER F+ 3SWAP 3OVER
 F- FABS EPSILON F<
 IF 3DUP 10 8 W.D. F.
 . " IN" I . ." ITERATIONS"
 EXIT THEN
 3SWAP I SFLOAT F* 3SWAP
LOOP 3DROP 3DROP ;
```

The algorithm used is:

```
 Prod = 2.0
 Oldsum = 2.0
 Do I = 3, 32767
 Newsum = Oldsum + (1/Prod)
 If (|Newsum - Oldsum| < Epsilon) EXIT
 Oldsum = Newsum
 Prod = Prod * I
 Continue
```

The upper limit on the DO loop (32,767) is set to the largest integer value to guarantee that the loop is executed *many* times. In actuality, the loop will terminate when the word EXIT is executed (when the newsum has satisfactorily converged to the oldsum). This exercise is a good example of stack manipulation and how it is hard to display in an algorithm the true efficient use of the stack.

## CHAPTER 15

```
14) : FACT DUP 0< OVER 7 > OR
 IF ." N IS OUT OF RANGE" DROP
 ELSE DUP 2 < IF 1 SWAP (VALUE OF 0! AND 1!)
 ELSE DUP BEGIN 1- DUP ROT *
 SWAP DUP 3 <
 END
 THEN
 DROP ." FACTORIAL =" .
 THEN ;
```

15) Because  $8! = 40,320$  which exceeds the range of a single-word integer.

16)

```
2.71828182 FCONSTANT E
3.14159265 FCONSTANT PI
: N! SFLOAT 3DUP 2.0 F* PI F* FSQRT
 3OVER 3DUP E F/ 3SWAP X**Y F*
 6 4 W.D 3DUP E. ." < "
 3OVER SFIX . ." ! < "
 3OVER 12.0 F* 1.0 F- 1.0 3SWAP F/
 1.0 F+ F* E. 3DROP ;
```

Compare the definitions of  $E$  and  $PI$  with the definition of  $1/3$  in exercise 7 of this chapter. Since there are no rational formulas for  $e$  and  $\pi$  we must specify each constant using as many digits of precision (9 decimal digits for KPN0 FORTH, Section 7.3) as provided by the floating-point data structure. If this definition were used on a computer with a different number of digits of precision then the definitions for  $e$  and  $\pi$  should be changed accordingly.

## CHAPTER 15

- 17) 0.0 REAL 1SUM ( SUM OF X[I] )  
 0.0 REAL 2SUM ( SUM OF X[I]\*\*2 )

```

: STAT 0.0 1SUM F! 0.0 2SUM F!
 ." HOW MANY NUMBERS? " SASK DUP
 0 DO CR ." ?" FASK 3DUP
 1SUM F@ F+ 1SUM F!
 3DUP F* 2SUM F@ F+ 2SUM F!
 LOOP
 DUP SFLOAT 1SUM F@ 3SWAP F/
 CR ." MEAN = " F.
 SFLOAT 3DUP 2SUM F@ F*
 1SUM F@ 3DUP F* F-
 3SWAP 3DUP 1.0 F- F* F/ FSQRT
 CR ." STD DEV = " F. ;

```

- 18) The shortest and most elegant (and possibly least obvious) solution is to rearrange the algorithm as follows:

```

 r = a
 → a = b
 b = r
 r = (a mod b)
 If (r = 0) then the answer is b, stop.
 └─ Loop around

```

This is coded as

```

: GCD BEGIN SWAP OVER MOD DUP 0= END
 DROP . ;

```

## CHAPTER 15

### 18) Continued

The rearranging of the algorithm places the test for  $r = 0$  at the end of the loop which allows the use of the BEGIN - END loop. You should confirm to yourself that rearranging the algorithm this way does not affect the algorithm (i.e. - it still produces the right answer).

$$\text{GCD}(2166, 6099) = 57$$

### 19)

```
2047 ()DIM DATA
 0 CONSTANT NOP
```

```
16105 0 DATA ! 18291 1 DATA ! 14333 2 DATA !
17015 3 DATA ! 15280 4 DATA ! 5 ' NOP !
```

```
: M+RMS 0.0 (RMS SUM) 0.0 (MEAN SUM) NOP 0
DO I DATA @ SFLOAT F+ (UPDATE MEAN) 3SWAP
 I DATA @ SFLOAT 3DUP F* F+ (RMS) 3SWAP
LOOP NOP SFLOAT F/ FROUND SFIX ." MEAN =" .
NOP SFLOAT F/ FSQRT FROUND SFIX ." , RMS =" . ;
```

### 20)

```
: RMEAN 0 (INITIAL MEAN) NOP 0
DO DUP I DATA @ SWAP -
 I 1 + / +
LOOP ." MEAN =" . ;
```



```

21) 0 VARIABLE C
 0 VARIABLE D
 0 VARIABLE E
 0 VARIABLE G
 0 VARIABLE N
 0 VARIABLE X
 0 VARIABLE Z
 0 VARIABLE YEAR

```

```

: NEASTER DUP DUP YEAR !
 19 MOD 1 + G !
 100 / 1 + DUP DUP C !
 3 * 4 / 12 - X !
 8 * 5 + 25 / 5 - Z !
 YEAR @ 5 * 4 / X @ - 10 - D !
 G @ 11 * 20 + Z @ + X @ - 30 MOD DUP
 0< IF 30 + THEN DUP DUP E !
 25 = G @ 1 > AND SWAP 24 = OR
 IF E @ 1 + E ! THEN
 44 E @ - DUP N ! 21 <
 IF N @ 30 + N ! THEN
 N @ DUP 7 + SWAP D @ + 7 MOD -
 DUP DUP N ! ; (RETURN 2 COPIES OF N ON STACK)

```

```

: EASTER NEASTER 31 >
 IF ." APRIL" 31 - .
 ELSE ." MARCH" . THEN
 ." ," YEAR @ . ;

```

## APPENDIX D - FORTH GLOSSARY

This glossary is an alphabetically ordered list of all standard KPN0 FORTH words along with a brief description of the word. The alphabetical ordering corresponds to the ordering of the ASCII character set (Appendix A). Additionally, a listing of the ASCII ordering is given at the top of each page for quick reference (since FORTH uses so many non-alphabetic characters).

Immediately following the name of a word, certain descriptor characters may appear within parentheses. These denote some special action or characteristics:

- A     The word belongs to the assembler vocabulary. A thorough description of the machine instructions is not given, instead the reader should refer to the Varian 620/f Computer Handbook.
- C     The word may be used only within a colon-definition. A following digit (C0 or C2) indicates the number of memory cells used when the word is compiled if other than one. A following + or - sign indicates that the word either pushes a value onto the stack or removes one from the stack during compilation. (This action is not related to its action during execution and may be implementation dependent.)
- E     The word may not normally be compiled within a colon-definition.
- P     The word has its precedence bit set; it is executed directly, even when encountered during compile mode.
- OLD   The word exists in KPN0 FORTH versions 2.3 and earlier.

Following the optional descriptor characters, a symbolic execution of the word is given, showing the parameters expected on the stack by the word and the result left on the stack (if any). The following notation is used:

- <ADDRESS>            denotes a 15-bit machine address;
- <BLOCK#>             denotes a FORTH block number;

|                 |                                                                                                                                                                  |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <BYTE-ADDRESS>  | denotes a 16-bit byte address;                                                                                                                                   |
| <CHAR-CODE>     | denotes a 7-bit integer value for a ASCII character<br>(see Appendix A);                                                                                         |
| <DW-VALUE>      | denotes a double-word integer value;                                                                                                                             |
| <FP-VALUE>      | denotes a floating-point value;                                                                                                                                  |
| <LINE#>         | denotes a line number of a FORTH block;                                                                                                                          |
| <LOGICAL-VALUE> | denotes a logical flag whereby a non-zero value<br>specifies true and a zero value specifies false;                                                              |
| <NAME>          | denotes a FORTH name, that is, a sequence of ASCII<br>characters whose first three characters and length<br>will be used to identify an entry in the dictionary; |
| <VALUE>         | denotes a single-word integer value.                                                                                                                             |

Any symbol that does not appear in the above list is a single-word integer value, unless the first two characters are DW (denoting a double-word integer value) or FP (denoting a floating-point value).

This list is purposely not broken down into vocabularies (basic FORTH, Utility words, etc.) in order that one be able to locate a word quickly, without having to search many different lists. It is expected that the greatest use of this list will be to aid someone who is going through a FORTH listing, in being able to quickly locate a description of a word they are not familiar with. Numerous lists are provided at the end of the glossary to provide a logical grouping of words with similar functions.

Since this list is not broken down by vocabularies, one should not expect to find all of these words defined in basic FORTH! In fact, only a small percentage of the words are defined in the basic FORTH system. In order to find just where on a FORTH tape a particular word is defined, simply obtain a cross reference of the tape (as with the XFORTH program, described in Appendix B of the "FORTH Systems Reference Manual") and from the cross reference find the block in which the word is defined. One is then able to

explicitly load the word into the dictionary. Naturally, this procedure may have to be gone through more than once if the desired word requires other words to be in the dictionary.

- !                   <VALUE> <ADDRESS> !  
STORE <VALUE> AT MEMORY LOCATION <ADDRESS>.
- !!                   <ADDRESS> !! <NAME> ... ;!C  
                  <ADDRESS> !! <NAME> ... ;!  
START AN INTERRUPT PROCESSING COLON DEFINITION (SIMILAR TO !).  
<ADDRESS> SPECIFIES THE LOW-CORE INTERRUPT VECTOR ADDRESS,  
DESIGNATING WHICH DEVICE'S INTERRUPTS ARE TO BE PROCESSED BY  
THIS WORD. THE DEFINITION IS TERMINATED BY EITHER ;!C OR ;!  
(SIMILAR TO ;). ;!C WILL POP THE CAMAC BRANCH DRIVER BEFORE  
RETURNING FROM THE INTERRUPT. SEE \$! AND CHAPTER 16.
- !BLOCK   (OLD)  
          RENAMED BUFFER.
- !I/O  
SAVES ALL THE SYSTEM FLAGS AND PARAMETERS THAT MUST BE SAVED  
PRIOR TO PERFORMING I/O FROM AN INTERRUPT WORD. THIS WORD  
INCLUDES THE EXECUTION OF FSAVE. AFTER PERFORMING THE I/O THE  
INTERRUPT WORD MUST EXECUTE @I/O TO RESTORE THESE FLAGS AND  
PARAMETERS.
- #           (A)  
SETS THE VARIABLE MODE TO 1, SPECIFYING AN IMMEDIATE OPERAND  
FOR THE NEXT MEMORY REFERENCE INSTRUCTION.
- #D  
A VARIABLE INDICATING THE NUMBER OF DIGITS APPEARING AFTER THE  
COMMA OR PERIOD, FOLLOWING AN INPUT NUMBER CONVERSION.
- #DEV       (OLD)  
          RENAMED #MDEV.
- #MDEV  
A CONSTANT WHOSE VALUE INDICATES THE PRIMARY MASS STORAGE  
DEVICE THAT FORTH IS RUNNING FROM:  
    0 = DISC  
    1 = TAPE
- #TER  
A CONSTANT WHOSE VALUE INDICATES WHAT TYPE OF TERMINAL IS BEING  
USED:  
    1 = TELETYPE  
    2 = TEKTRONIX 4010  
    4 = TEC  
    6 = LEAR-SIEGLER ADM-3A  
    7 = TEXAS INSTRUMENTS TI-700

**\$!**                   <MODULE-ID> \$! <ADDRESS>  
 CONVERTS THE CAMAC <MODULE-ID> INTO THE LOW-CORE INTERRUPT VECTOR <ADDRESS> FOR THAT MODULE (REFER TO BLOCK 54 FOR A LISTING OF THE STANDARD KPNO MODULE IDENTIFIERS). SEE !! AND CHAPTER 16.

**\$2AREAD;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$2AWRITE;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$2READ;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$2WRITE;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$ACOMMAND;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$AREAD;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$AWRITE;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$C**  
 A CAMAC WORD TO SEND A CLEAR COMMAND TO THE MODULES IN CRATE 1.

**\$CN;**                   <VALUE> \$CN; <NAME>  
 DEFINE <NAME> AS A CAMAC <MODULE-ID>. SEE CHAPTER 16.

**\$COMMAND;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.

**\$DBD**  
 A CAMAC WORD TO DISABLE BRANCH DEMANDS AT THE CRATE CONTROLLER LEVEL. SEE \$EBD.

**\$DIR**  
 A CAMAC WORD TO DISABLE INTERRUPTS AT THE BRANCH DRIVER LEVEL. SEE \$EIR.

**\$EBD**  
 A CAMAC WORD TO ENABLE BRANCH DEMANDS AT THE CRATE CONTROLLER LEVEL. SEE \$DBD.

**\$EIR**  
 A CAMAC WORD TO ENABLE CAMAC INTERRUPTS AT THE BRANCH DRIVER LEVEL. SEE \$DIR.

- \$FX**                   <VALUE> \$FX <RESULT>  
 <VALUE> MUST BE A CAMAC F CODE IN THE RANGE 0 THROUGH 31 AND THIS VALUE IS THEN CONVERTED TO THE APPROPRIATE EXC INSTRUCTION (FOR USE IN A SEQUENCE OF MACHINE INSTRUCTIONS). THE WORD \$FX IS USUALLY FOLLOWED BY THE WORD , WHICH WILL PLACE THE EXC INSTRUCTION INTO THE NEXT AVAILABLE DICTIONARY LOCATION.
- \$INITIALIZE**  
 A CAMAC WORD TO INITIALIZE THE BRANCH DRIVER.
- \$NOI**  
 A CAMAC WORD TO CLEAR THE INHIBIT FLIP-FLOP IN THE CRATE CONTROLLER.
- \$READ;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.
- \$REPLACE**            \$REPLACE <WORD1> <CHAR-STRING>\$  
 REPLACE ALL OCCURENCES OF <WORD1> BY THE SPECIFIED <CHAR-STRING> WHEN THE WORD FIX IS EXECUTED. <WORD1> MAY NOT CONTAIN ANY SPACES. <CHAR-STRING> STARTS WITH THE SECOND CHARACTER FOLLOWING <WORD1> (THE FIRST CHARACTER FOLLOWING <WORD1> MUST BE THE SPACE THAT TERMINATES <WORD1>) AND INCLUDES ALL CHARACTERS, INCLUDING SPACES, UP TO BUT NOT INCLUDING THE DOLLAR SIGN. SEE REPLACE, WINIT AND FIX.
- \$SETUP**  
 A CAMAC WORD TO INITIALIZE AND RESET THE CAMAC SYSTEM.
- \$WRITE;**  
 DEFINE A CAMAC I/O WORD. SEE CHAPTER 16.
- \$Z**  
 A CAMAC WORD TO SEND AN INITIALIZE COMMAND TO CRATE 1.
- &**                    &<CHARACTER> <CHAR-CODE>  
 THE AMPERSAND CONVERTS THE <CHARACTER> IMMEDIATELY FOLLOWING IT TO ITS 7-BIT, ASCII CODE (AN INTEGER VALUE IN THE RANGE 0 THRU 127). FOR EXAMPLE, THE SEQUENCE "&A" WILL LEAVE THE OCTAL VALUE 101 ON THE STACK. REFER TO APPENDIX A FOR A COMPLETE LISTING OF ALL ASCII CODES.
- '**                    ' <NAME> <ADDRESS>  
 PUSH THE ADDRESS OF THE PARAMETER FIELD OF <NAME> ONTO THE STACK. A COMPILER DIRECTIVE, ' IS EXECUTED WHEN ENCOUNTERED IN A COLON DEFINITION: THE ADDRESS OF THE PARAMETER FIELD OF <NAME> IS FOUND IMMEDIATELY (AT COMPILATION) AND STORED IN THE DICTIONARY (AFTER THE ADDRESS OF /LIT/) AS A LITERAL TO BE PLACED ON THE STACK AT EXECUTION TIME. WITHIN A COLON DEFINITION, THE SEQUENCE "' <NAME>" IS IDENTICAL TO THE SEQUENCE "/LIT/ [ ' <NAME> , ]".

( (P) ( <STRING> )  
 THE LEFT PAREN DESIGNATES THE START OF A COMMENT AND ALL CHARACTERS UP TO THE RIGHT PAREN ARE IGNORED. SINCE ( IS A FORTH WORD IT MUST BE TERMINATED BY A SPACE, HOWEVER, THE CLOSING PAREN NEED NOT BE PRECEDED BY A SPACE. UP TO 1023 CHARACTERS MAY COMPRISE THE COMMENT.

( )DIM <VALUE> ( )DIM <NAME>  
 DEFINES A VECTOR OF SINGLE-WORD INTEGER VALUES. <VALUE> + 1 CELLS OF MEMORY ARE ALLOCATED TO THE NAMED VECTOR AND THEN LEGITIMATE INDICES WILL BE IN THE RANGE 0 THROUGH <VALUE>, INCLUSIVE. EXECUTING THE SEQUENCE "<INDEX> <NAME>" WILL PUSH ONTO THE STACK THE ADDRESS OF THE SPECIFIED ENTRY IN THE VECTOR.

\* <VALUE1> <VALUE2> \* <RESULT>  
 16-BIT, SIGNED, INTEGER MULTIPLY, LEAVING THE SINGLE-WORD RESULT ON THE STACK.

\*\* <VALUE> <POWER> \*\* <RESULT>  
 INTEGER EXPONENTIATION. RAISE <VALUE> TO THE SPECIFIED <POWER> AND LEAVE THE SINGLE-WORD INTEGER RESULT ON THE STACK.

( \*, <VALUE1> <VALUE2> \*, <RESULT>  
 MULTIPLY 14-BIT FRACTIONS. IF <VALUE1> AND <VALUE2> ARE 14-BIT FRACTIONS IN THE RANGE -2.000 TO 1.9999 THEN THE RESULT WILL ALSO BE A 14-BIT FRACTION IN THIS RANGE.

\*/ <VALUE1> <VALUE2> <VALUE3> \*/ <RESULT>  
 CALCULATE <VALUE1> \* <VALUE2> / <VALUE3> AND LEAVE THE RESULT ON THE STACK. THE INTERMEDIATE RESULT FROM THE MULTIPLICATION IS 31-BITS AND THIS WORD THEREFORE PROVIDES GREATER ACCURACY THAN THE SEQUENCE "<VALUE1> <VALUE2> \* <VALUE3> /". NOTE THAT THE DIVISION IS AN INTEGER DIVISION WITH TRUNCATION AND ANY REMAINDER IS LOST.

\*10\*\* <FP-VALUE> <POWER> \*10\*\* <FP-RESULT>  
 MULTIPLY THE <FP-VALUE> BY THE SPECIFIED INTEGER POWER OF 10, LEAVING THE FLOATING-POINT RESULT ON THE STACK.

\*BLOCK (OLD)  
 RENAMED +BLOCK.



- +**                   <VALUE1> <VALUE2> + <RESULT>  
16-BIT SIGNED INTEGER ADDITION, LEAVING THE RESULT ON THE STACK.
- +!**                   <VALUE> <ADDRESS> +!  
ADD <VALUE> TO THE CURRENT CONTENTS OF THE MEMORY LOCATION POINTED TO BY <ADDRESS>. <VALUE> MAY BE A POSITIVE OR NEGATIVE NUMBER. IDENTICAL TO THE SEQUENCE: "<ADDRESS> @ <VALUE> + <ADDRESS> !".
- +BLOCK**            <VALUE> +BLOCK <BLOCK#>  
ADD <VALUE> TO THE NUMBER OF THE CURRENT BLOCK BEING INTERPRETED AND LEAVE THE RESULT ON THE STACK. FOR EXAMPLE, IN BLOCK 350 THE SEQUENCE "2 +BLOCK" WILL LEAVE THE NUMBER 352 ON THE STACK.
- +CONVERT**           <VALUE> <DW-VALUE> +CONVERT <COUNT>  
CONVERTS THE <DW-VALUE> INTO ITS SEQUENCE OF ASCII CHARACTERS FOR OUTPUT BY THE WORDS WRITE OR TYPE. THE CURRENT NUMBER CONVERSION BASE IS USED. <DW-VALUE> MUST BE A POSITIVE NUMBER AND <VALUE> IS THEN USED TO SPECIFY THE SIGN: IF <VALUE> IS NEGATIVE A MINUS SIGN WILL PRECEDE THE NUMBER. ON RETURN THE BYTE ADDRESS OF THE CHARACTER STRING IS CONTAINED IN IP AND THE CHARACTER COUNT IS ON TOP OF THE STACK. THE VARIABLES FLD AND DPL ARE USED TO SPECIFY THE TOTAL FIELD WIDTH AND NUMBER OF DIGITS TO THE RIGHT OF THE RADIX POINT.
- +LOOP**            (C)    <VALUE> +LOOP  
ADD <VALUE> TO THE CURRENT LOOP INDEX (REFER TO THE WORDS DO AND LOOP). EXIT FROM THE LOOP IS MADE WHEN THE RESULTANT INDEX REACHES OR PASSES THE LIMIT IF <VALUE> IS POSITIVE, OR WHEN THE INDEX IS LESS THAN THE LIMIT IF <VALUE> IS NEGATIVE.
- ,**                   <VALUE> ,  
STORE <VALUE> INTO THE NEXT AVAILABLE DICTIONARY CELL, ADVANCING THE DICTIONARY POINTER.

- <VALUE1> <VALUE2> - <RESULT>  
16-BIT SIGNED INTEGER SUBTRACTION LEAVING THE RESULT, <VALUE1>  
- <VALUE2>, ON THE STACK.
  
- >           (P)           -->  
CONTINUE INTERPRETATION WITH THE NEXT BLOCK. THIS WORD IS  
SIMILAR TO THE SEQUENCE "1 +BLOCK CONTINUED", HOWEVER --> IS A  
COMPILER DIRECTIVE AND IS THEREFORE ESPECIALLY USEFUL WHEN  
EXTENDING A COLON DEFINITION FROM ONE BLOCK TO THE NEXT.
  
- CONVERT (OLD)  
CLEANS UP THE STACK AFTER EXECUTING THE OLD VERSIONS OF  
+CONVERT AND WRITE.
  
- INR,           (A)           <ADDRESS> -INR,  
AN ASSEMBLER MACRO WHICH GENERATES A SEQUENCE OF MACHINE  
INSTRUCTIONS TO DECREMENT THE CONTENTS OF THE SPECIFIED MEMORY  
ADDRESS. THE SEQUENCE OF INSTRUCTIONS GENERATED IS AN LDA, DAR  
AND STA.
  
- .                   <VALUE> .  
PRINT <VALUE> ON THE CURRENT OUTPUT DEVICE (USUALLY THE  
OPERATOR'S TERMINAL), FREE FORMAT, CONVERTED ACCORDING TO THE  
CURRENT NUMBER BASE.
  
- ."                ." <STRING>"  
OUTPUT THE CHARACTER STRING TO THE CURRENT OUTPUT DEVICE  
(USUALLY THE OPERATOR'S TERMINAL). <STRING> STARTS WITH THE  
SECOND CHARACTER FOLLOWING ." (THE FIRST CHARACTER FOLLOWING ."   
MUST BE A SPACE). THE MAXIMUM NUMBER OF CHARACTERS THAT MAY  
COMPRISE <STRING> IS 127.
  
- .FIX                <FP-VALUE> .FIX <DW-FRACTION-RESULT>  
CONVERT THE <FP-VALUE> TO A DOUBLE-WORD FRACTION. TRUNCATION  
WILL OCCUR IF THE ABSOLUTE VALUE OF <FP-VALUE> IS GREATER THAN  
OR EQUAL TO 1.0 AND A RESULT OF ZERO WILL BE RETURNED IF THE  
ABSOLUTE VALUE OF <FP-VALUE> IS TOO SMALL (< 2\*\*-31).
  
- .FLOAT            <DW-FRACTION> .FLOAT <FP-RESULT>  
<DW-FRACTION> IS CONVERTED TO A FLOATING-POINT VALUE.
  
- .STRING            <ADDRESS> .STRING  
EQUIVALENT TO THE SEQUENCE "COUNT WRITE", HOWEVER, .STRING MAY  
BE EXECUTED FROM THE TERMINAL TO OUTPUT A STRING WHEREAS THE  
SEQUENCE "COUNT WRITE" MAY NOT (SINCE IP, WHICH IS SET BY  
COUNT, MUST REMAIN INTACT FOR WRITE, AND IP IS CHANGED EACH  
TIME A WORD IN A LINE OF TERMINAL INPUT IS PROCESSED).

- /  $\langle \text{VALUE1} \rangle \langle \text{VALUE2} \rangle / \langle \text{RESULT} \rangle$   
 16-BIT SIGNED INTEGER DIVIDE. THE RESULT,  $\langle \text{VALUE1} \rangle / \langle \text{VALUE2} \rangle$  IS LEFT ON THE STACK. NOTE THAT THE QUOTIENT IS TRUNCATED AND ANY REMAINDER IS LOST. SEE /MOD.
- /,  $\langle \text{VALUE1} \rangle \langle \text{VALUE2} \rangle /, \langle \text{RESULT} \rangle$   
 DIVIDE  $\langle \text{VALUE1} \rangle$  BY  $\langle \text{VALUE2} \rangle$  LEAVING THE RESULT ON THE STACK. IF BOTH DIVIDEND AND DIVISOR ARE 14-BIT FRACTIONS IN THE RANGE -2.000 TO 1.9999 THEN THE QUOTIENT WILL ALSO BE A 14-BIT FRACTION IN THE SAME RANGE. SEE \*,.
- /4010  
 A 2VARIABLE WHOSE VALUE INDICATES THE CURRENT PHYSICAL POSITION OF THE 4010.
- /CURSE  
 A 2VARIABLE WHOSE VALUE INDICATES THE CURRENT PHYSICAL POSITION OF THE 4010 CROSS HAIR CURSORS.
- /LIT/ (C) /LIT/  
 A REFERENCE TO /LIT/ IS AUTOMATICALLY COMPILED BEFORE EACH LITERAL ENCOUNTERED IN A COLON DEFINITION. EXECUTION OF /LIT/ CAUSES THE CONTENTS OF THE NEXT DICTIONARY CELL TO BE PUSHED ONTO THE STACK.
- /MOD  $\langle \text{VALUE1} \rangle \langle \text{VALUE2} \rangle /MOD \langle \text{REMAINDER} \rangle \langle \text{QUOTIENT} \rangle$   
 16-BIT SIGNED INTEGER DIVIDE. THE QUOTIENT FROM THE DIVISION,  $\langle \text{VALUE1} \rangle / \langle \text{VALUE2} \rangle$ , IS LEFT ON TOP OF THE STACK AND THE REMAINDER IS LEFT BELOW. THE REMAINDER HAS THE SIGN OF THE DIVIDEND.

0) (A) <ADDRESS> 0) <RESULT>  
 THE MOST SIGNIFICANT BIT OF <ADDRESS> IS SET TO 1, CHANGING THE ADDRESS TO AN INDIRECT ADDRESS. NOTE THAT THIS WORD DESIGNATES AN ADDRESS AS INDIRECT WHILE THE WORD I) DESIGNATES AN INSTRUCTION AS INDIRECT.

0)S <ADDRESS>  
 0)S IS THE STARTING ADDRESS OF A VECTOR OF ADDRESSES AND INTERNAL VALUES USED BY FORTH. THE SEQUENCE "0)S <VALUE> +" FORMS THE ADDRESS OF A SPECIFIC ELEMENT IN THE VECTOR WITH <VALUE> CORRESPONDING TO:

- 0 THE ADDRESS OF THE CHARACTER OUTPUT SUBROUTINE. THIS SUBROUTINE MUST BE CALLED USING AN IJMP INSTRUCTION.
- 1 THE ADDRESS OF THE CHARACTER INPUT SUBROUTINE. THIS SUBROUTINE MUST BE CALLED USING AN IJMP INSTRUCTION.
- 2 THE ADDRESS OF THE TERMINAL INTERROGATION SUBROUTINE. THIS SUBROUTINE MUST BE CALLED USING AN IJMP INSTRUCTION.
- 3 THE ADDRESS OF THE ROUTINE FETCH. THIS SUBROUTINE MUST BE CALLED USING AN IJMP INSTRUCTION AND ON RETURN THE A REGISTER CONTAINS THE CHARACTER WHICH WAS POINTED TO BY THE BYTE-ADDRESS IN IP. IP IS INCREMENTED.
- 4 THE ADDRESS OF THE SUBROUTINE DEPOSIT. THIS SUBROUTINE MUST BE CALLED USING AN IJMP INSTRUCTION. ON ENTRY THE A REGISTER MUST CONTAIN THE CHARACTER TO BE STORED IN THE LOCATION POINTED TO BY THE BYTE ADDRESS IN OP. OP IS INCREMENTED.
- 5 THE CURRENT CORE-SIZE USED BY FORTH. THE INITIAL VALUE OF 8192 (20000B) SPECIFIES 8K WORDS OF CORE (ADDRESSES 0 THRU 8191).
- 6 A BLOCK NUMBER OFFSET THAT IS ADDED TO EVERY BLOCK NUMBER WHEN REFERENCING A DISC BLOCK. NORMALLY ZERO.
- 7 A POINTER TO THE PSEUDO-WORD USED FOR FLOATING-POINT NUMBER CONVERSIONS. IT MUST BE ZERO IF THE FLOATING-POINT CODE IS NOT IN THE DICTIONARY.
- 8 THE WORD COUNT USED FOR DATA TRANSFERS TO THE DISC, NORMALLY 512.

0< <VALUE> 0< <LOGICAL-VALUE>  
 0<= <VALUE> 0<= <LOGICAL-VALUE>  
 0<> <VALUE> 0<> <LOGICAL-VALUE>  
 0= <VALUE> 0= <LOGICAL-VALUE>  
 0> <VALUE> 0> <LOGICAL-VALUE>  
 0>= <VALUE> 0>= <LOGICAL-VALUE>

COMPARE <VALUE> AGAINST ZERO AND LEAVE A <LOGICAL-VALUE> OF TRUE ON THE STACK IF THE INDICATED RELATION IS TRUE, OTHERWISE A <LOGICAL-VALUE> OF FALSE IS LEFT ON THE STACK. THE WORD 0<> TESTS FOR NOT EQUAL TO ZERO.

- 1+                   <VALUE> 1+ <RESULT>  
EQUIVALENT TO THE SEQUENCE "1 +".
- 1+!                  <ADDRESS> 1+!  
ADD 1 TO THE CONTENTS OF THE MEMORY LOCATION <ADDRESS>.  
EQUIVALENT TO THE SEQUENCE "<ADDRESS> @ 1+ <ADDRESS> !".
- 1-                   <VALUE> 1- <RESULT>  
EQUIVALENT TO THE SEQUENCE "1 -".
- 1-!                  <ADDRESS> 1-!  
SUBTRACT 1 FROM THE CONTENTS OF MEMORY LOCATION <ADDRESS>.  
EQUIVALENT TO THE SEQUENCE "<ADDRESS> @ 1- <ADDRESS> !".
- 1LRL                (OLD)  
RENAMED BYTE.
- 2()DIM              <VALUE> 2()DIM <NAME>  
DEFINES A VECTOR OF DOUBLE-WORD VALUES. <VALUE> + 1 DOUBLE-WORD  
CELLS OF MEMORY ARE ALLOCATED TO THE NAMED VECTOR AND THEN  
LEGITIMATE INDICES WILL BE IN THE RANGE 0 THROUGH <VALUE>,  
INCLUSIVE. EXECUTING THE SEQUENCE "<INDEX> <NAME>" WILL PUSH  
ONTO THE STACK THE ADDRESS OF THE SPECIFIED ENTRY IN THE  
VECTOR.
- 2\*                   <VALUE> 2\* <RESULT>  
EQUIVALENT TO THE SEQUENCE "2 \*".
- 2/                   <VALUE> 2/ <RESULT>  
EQUIVALENT TO THE SEQUENCE "2 /".
- 2CONSTANT           <DW-VALUE> 2CONSTANT <NAME>  
DEFINE THE WORD <NAME> WHICH WHEN EXECUTED WILL PUSH ONTO THE  
STACK ITS DOUBLE-WORD VALUE. THE VALUE OF <NAME> IS INITIALIZED  
TO <DW-VALUE>. THE VALUE OF <NAME> MAY BE CHANGED BY EXECUTING  
THE SEQUENCE "<DW-VALUE> ' <NAME> D!".
- 2DROP                <DW-VALUE> 2DROP  
DROP THE TOP TWO VALUES FROM THE STACK. THE TOP TWO VALUES MAY  
BE TWO SINGLE-WORD VALUES OR (USUALLY) A DOUBLE-WORD VALUE.
- 2DUP                 <DW-VALUE> 2DUP <DW-VALUE> <DW-VALUE>  
DUPLICATE THE TOP TWO VALUES ON THE STACK. THE TOP TWO VALUES  
MAY BE TWO SINGLE-WORD VALUES OR (USUALLY) A DOUBLE-WORD VALUE.
- 2LS                  <DW-VALUE> <SHIFT-COUNT> 2LS <DW-RESULT>  
ROTATE THE <DW-VALUE> LEFT OR RIGHT. IF THE <SHIFT-COUNT> IS  
POSITIVE THE SHIFT IS A LOGICAL ROTATE LEFT WHILE IF THE  
<SHIFT-COUNT> IS NEGATIVE THEN THE SHIFT IS A LOGICAL ROTATE  
RIGHT.

- 2M\***                   <DW-VALUE> <VALUE> 2M\* <DW-RESULT>  
MULTIPLY THE SINGLE-WORD VALUE ON TOP OF THE STACK BY THE DOUBLE-WORD VALUE BELOW, LEAVING THE DOUBLE-WORD RESULT ON TOP OF THE STACK. SEE M\*.
- 2OVER**                <DW-VALUE1> <DW-VALUE2> 2OVER  
                          <DW-VALUE1> <DW-VALUE2> <DW-VALUE1>  
PUSH A COPY OF <DW-VALUE1> ONTO THE TOP OF THE STACK, WITHOUT REMOVING ANY WORDS FROM THE STACK.
- 2PICK**                <INDEX> 2PICK <DW-VALUE>  
<INDEX> SPECIFIES A LOCATION ON THE STACK (1 SPECIFIES THE TOP OF THE STACK, 2 IS THE NEXT CELL ON THE STACK, ETC) AND A COPY OF THE DOUBLE-WORD VALUE STARTING AT THIS LOCATION IS PUSHED ONTO THE TOP OF THE STACK. THE SEQUENCE "3 2PICK" IS EQUIVALENT TO 2OVER.
- 2ROLL**                <INDEX> 2ROLL  
<INDEX> SPECIFIES A DOUBLE-WORD POSITION ON THE STACK (1 SPECIFIES THE DOUBLE-WORD INTEGER ON TOP OF THE STACK, 2 THE DOUBLE-WORD INTEGER BELOW, ETC) AND THIS DOUBLE-WORD VALUE IS MOVED TO THE TOP OF THE STACK WITH ALL DOUBLE-WORD VALUES IN BETWEEN BEING MOVED DOWN ONE POSITION.
- 2ROT**                <DW-VALUE1> <DW-VALUE2> <DW-VALUE3> 2ROT  
                          <DW-VALUE2> <DW-VALUE3> <DW-VALUE1>  
ROTATE THE TOP THREE DOUBLE-WORD VALUES ON THE STACK. <DW-VALUE1> IS MOVED TO THE TOP OF THE STACK, <DW-VALUE3> MOVES FROM THE TOP TO THE SECOND POSITION AND <DW-VALUE2> MOVES FROM THE SECOND POSITION TO THE THIRD.
- 2SET**                <VALUE1> <ADDRESS1> <VALUE2> <ADDRESS2> 2SET <NAME>  
DEFINES THE WORD <NAME> WHICH, WHEN EXECUTED, WILL STORE <VALUE1> AT <ADDRESS1> AND <VALUE2> AT <ADDRESS2>.
- 2SWAP**                <DW-VALUE1> <DW-VALUE2> 2SWAP  
                          <DW-VALUE2> <DW-VALUE1>  
SWAP THE TWO DOUBLE-WORD VALUES ON TOP OF THE STACK.
- 2VARIABLE**           <DW-VALUE> 2VARIABLE <NAME>  
DEFINE THE WORD <NAME> WHICH, WHEN EXECUTED, WILL PUSH ONTO THE STACK THE ADDRESS OF <NAME>'S VALUE. THE VALUE OF THE VARIABLE IS INITIALIZED TO <DW-VALUE>. THE SEQUENCE "<NAME> D@" WILL PUSH THE VALUE OF THE VARIABLE ONTO THE STACK AND THE SEQUENCE "<DW-VALUE> <NAME> D!" WILL STORE <DW-VALUE> AS THE VARIABLE'S NEW VALUE.

- 3()DIM**                   <VALUE> 3()DIM <NAME>  
 DEFINES A VECTOR OF FLOATING-POINT VALUES. <VALUE> + 1  
 FLOATING-POINT CELLS OF MEMORY ARE ALLOCATED TO THE NAMED  
 VECTOR AND THEN LEGITIMATE INDICES WILL BE IN THE RANGE 0  
 THROUGH <VALUE>, INCLUSIVE. EXECUTING THE SEQUENCE "<INDEX>  
 <NAME>" WILL PUSH ONTO THE STACK THE ADDRESS OF THE SPECIFIED  
 ENTRY IN THE VECTOR.
- 3DROP**                   <FP-VALUE> 3DROP  
 DELETE THE TOP THREE VALUES FROM THE STACK. THE TOP THREE  
 VALUES USUALLY COMPRISE A SINGLE FLOATING-POINT NUMBER BUT MAY  
 ALSO CONSIST OF THREE SINGLE-WORD VALUES OR A DOUBLE-WORD VALUE  
 AND A SINGLE-WORD VALUE.
- 3DUP**                   <FP-VALUE> 3DUP <FP-VALUE> <FP-VALUE>  
 DUPLICATE THE TOP THREE VALUES ON THE STACK. THE TOP THREE  
 VALUES USUALLY COMPRISE A SINGLE FLOATING-POINT NUMBER BUT MAY  
 ALSO CONSIST OF THREE SINGLE-WORD VALUES OR A DOUBLE-WORD VALUE  
 AND A SINGLE-WORD VALUE.
- 3OVER**                   <FP-VALUE1> <FP-VALUE2> 3OVER  
                           <FP-VALUE1> <FP-VALUE2> <FP-VALUE1>  
 PUSH A COPY OF <FP-VALUE1> ONTO THE TOP OF THE STACK, WITHOUT  
 REMOVING ANY WORDS FROM THE STACK.
- 3PICK**                   <INDEX> 3PICK <FP-VALUE>  
 <INDEX> SPECIFIES A LOCATION ON THE STACK (1 SPECIFIES THE TOP  
 OF THE STACK, 2 IS THE NEXT CELL ON THE STACK, ETC) AND A COPY  
 OF THE FLOATING-POINT VALUE STARTING AT THIS LOCATION IS PUSHED  
 ONTO THE TOP OF THE STACK. THE SEQUENCE "4 3PICK" IS EQUIVALENT  
 TO 3OVER.
- 3ROLL**                   <INDEX> 3ROLL  
 <INDEX> SPECIFIES A FLOATING-POINT POSITION ON THE STACK (1  
 SPECIFIES THE FLOATING-POINT VALUE ON TOP OF THE STACK, 2 THE  
 FLOATING-POINT VALUE BELOW, ETC) AND THIS FLOATING-POINT VALUE  
 IS MOVED TO THE TOP OF THE STACK WITH ALL FLOATING-POINT VALUES  
 IN BETWEEN BEING MOVED DOWN ONE POSITION.
- 3ROT**                   <FP-VALUE1> <FP-VALUE2> <FP-VALUE3> 3ROT  
                           <FP-VALUE2> <FP-VALUE3> <FP-VALUE1>  
 ROTATE THE TOP THREE FLOATING-POINT VALUES ON THE STACK.  
 <FP-VALUE1> IS MOVED TO THE TOP OF THE STACK, <FP-VALUE3> MOVES  
 FROM THE TOP TO THE SECOND POSITION AND <FP-VALUE2> MOVES FROM  
 THE SECOND POSITION TO THE THIRD.
- 3SWAP**                   <FP-VALUE1> <FP-VALUE2> 3SWAP  
                           <FP-VALUE2> <FP-VALUE1>  
 SWAP THE TWO FLOATING-POINT VALUES ON TOP OF THE STACK.

- :** **<NAME> ... ;**  
 START A COLON DEFINITION, THAT IS CREATE A DICTIONARY ENTRY THAT WILL DEFINE <NAME> AS EQUIVALENT TO THE SEQUENCE OF WORDS BETWEEN <NAME> AND THE SEMICOLON. THE COMPILATION MODE FLAG IS SET AND THE CONTEXT VOCABULARY IS SET TO THE CURRENT VOCABULARY. THE COLON DEFINITION IS TERMINATED BY THE SEMICOLON.
- :ORX** **:ORX ... ;**  
 INITIATES AN ANONYMOUS (ORPHAN) COLON DEFINITION, PLACING ITS ADDRESS IN PSEUDO-VECTOR ENTRY X, FOR SUBSEQUENT ADOPTION BY THE WORD ADOX. AN ANONYMOUS COLON DEFINITION IS SIMILAR TO A STANDARD COLON DEFINITION, HOWEVER, IT MAY BE EXECUTED ONLY FROM A COLON DEFINITION (USING ADOX) NOT FROM THE TERMINAL AND THREE MEMORY CELLS ARE SAVED SINCE THE ANONYMOUS DEFINITION HAS NO NAME. SEE ADOX AND P-VX.
- ;** **(C,P)**  
 TERMINATE A COLON DEFINITION AND RESET THE COMPILATION MODE FLAG.
- ::** **(C,P) : <NAME1> ... :: ... ;**  
 TERMINATE A DEFINING WORD <NAME1>. THE DEFINING WORD <NAME1> CAN SUBSEQUENTLY BE EXECUTED TO DEFINE A NEW WORD, <NAME2>. WHEN <NAME2> IS EXECUTED IT WILL CAUSE THE WORDS BETWEEN :: AND ; TO BE EXECUTED WITH THE CONTENTS OF THE FIRST PARAMETER OF <NAME2> ON THE STACK.
- ::S** **(C) ;S**  
 THIS WORD MAY BE USED TO TERMINATE A COLON DEFINITION IN PLACE OF ;. WHEN THE COLON DEFINITION TERMINATED BY ::S COMPLETES EXECUTION (I.E. - THE WORD ;S IS EXECUTED) THE LOADING OF THE CURRENT BLOCK WILL TERMINATE AS IF THE WORD ;S WERE EXECUTED. THIS WORD IS NOT, IN GENERAL, EQUIVALENT TO THE SEQUENCE " ;S".
- ;CODE** **(C,P) : <NAME1> ... ;CODE ...**  
 STOP COMPILATION AND TERMINATE A DEFINING WORD, <NAME1>. THE CONTEXT VOCABULARY IS SET TO THE ASSEMBLER VOCABULARY. WHEN <NAME1> IS EXECUTED TO DEFINE A NEW WORD <NAME2>, THE EXECUTION ADDRESS OF <NAME2> WILL POINT TO THE ASSEMBLER CODE SEQUENCE FOLLOWING THE ;CODE OF <NAME1>. SUBSEQUENT EXECUTION OF <NAME2> WILL CAUSE THIS MACHINE CODE SEQUENCE TO BE EXECUTED.
- ;S** **(E)**  
 STOP INTERPRETATION OF A SYMBOLIC BLOCK.



|    |          |          |    |                 |
|----|----------|----------|----|-----------------|
| <  | <VALUE1> | <VALUE2> | <  | <LOGICAL-VALUE> |
| <= | <VALUE1> | <VALUE2> | <= | <LOGICAL-VALUE> |
| <> | <VALUE1> | <VALUE2> | <> | <LOGICAL-VALUE> |
| =  | <VALUE1> | <VALUE2> | =  | <LOGICAL-VALUE> |
| >  | <VALUE1> | <VALUE2> | >  | <LOGICAL-VALUE> |
| >= | <VALUE1> | <VALUE2> | >= | <LOGICAL-VALUE> |

COMPARE <VALUE1> AND <VALUE2> AND LEAVE A <LOGICAL-VALUE> OF TRUE ON THE STACK IF THE INDICATED RELATION IS TRUE, OTHERWISE LEAVE A <LOGICAL-VALUE> OF FALSE ON THE STACK. THE WORD <> TESTS FOR NOT EQUAL.

<<LX (A) <ADDRESS> <<LX  
FIX A MEMORY REFERENCE INSTRUCTION'S RELATIVE ADDRESS. SEE >>LX.

<T> (A)  
A CONSTANT WHOSE VALUE IS THE ADDRESS OF A CORE LOCATION AVAILABLE FOR TEMPORARY STORAGE. THE CONTENTS OF THIS CORE LOCATION WILL BE SAVED DURING INTERRUPT PROCESSING.

=2 (A) =2 <ADDRESS>  
PUSHES ONTO THE STACK THE STARTING ADDRESS OF A VECTOR OF BINARY CONSTANTS USED INTERNALLY BY FORTH. DO NOT CHANGE THESE VALUES. THESE VALUES ARE ALL IN LOW CORE AND ARE THEREFORE USEFUL IN MACHINE LANGUAGE WORDS WHEN THE GIVEN CONSTANT NEEDS TO BE REFERENCED BY A SINGLE WORD INSTRUCTION. THE CONSTANTS AND HOW TO ACCESS THEM IS AS FOLLOWS:

|    |    |   |   |                             |
|----|----|---|---|-----------------------------|
| =2 | 2  | - | @ | LEAVES 100000B ON THE STACK |
| =2 | 1- | @ |   | LEAVES 077777B ON THE STACK |
| =2 | @  |   |   | LEAVES 2 ON THE STACK       |
| =2 | 1+ | @ |   | LEAVES 3 ON THE STACK       |
| =2 | 2  | + | @ | LEAVES 4 ON THE STACK       |
| =2 | 3  | + | @ | LEAVES 5 ON THE STACK       |
| =2 | 4  | + | @ | LEAVES 6 ON THE STACK       |
| =2 | 5  | + | @ | LEAVES 7 ON THE STACK       |

&gt;&gt;LX

(A)

A WORD USED IN GENERATING MACHINE LANGUAGE FORWARD REFERENCES. THE CURRENT VALUE OF THE DICTIONARY POINTER IS STORED IN THE CORRESPONDING PSEUDO-VECTOR TABLE ENTRY (SEE P-VX) AND A VALUE OF ZERO IS PUSHED ONTO THE STACK. THE ADDRESS THAT IS SAVED IN THE PSEUDO-VECTOR TABLE IS ASSUMED TO POINT TO A VARIAN MEMORY REFERENCE INSTRUCTION. THE VALUE OF ZERO THAT IS PUSHED ONTO THE STACK WILL THEN BE USED AS THE MEMORY ADDRESS OF THE MEMORY REFERENCE INSTRUCTION, GUARANTEEING THAT THE SINGLE-WORD VERSION OF THE INSTRUCTION WILL BE GENERATED. THE WORD <<LX MUST THEN BE EXECUTED LATER IN THE COMPILATION TO CHANGE THE MEMORY REFERENCE INSTRUCTION TO AN ADDRESSING MODE OF 4 (RELATIVE TO THE P REGISTER) AND TO SET THE RELATIVE ADDRESS OF THE INSTRUCTION TO THE DIFFERENCE OF THE SPECIFIED ADDRESS (THE <ADDRESS> PUSHED ONTO THE STACK BEFORE <<LX IS EXECUTED) AND THE MEMORY ADDRESS THAT >>LX STORED IN THE PSEUDO VECTOR TABLE. THIS LENGTHY PROCEDURE IS REQUIRED TO GENERATE FORWARD REFERENCES USING FORTH'S SINGLE PASS STRUCTURE AND THE VARIAN MACHINE INSTRUCTIONS.

&gt;BCD

A VARIABLE WHOSE VALUE SPECIFIES WHETHER THE 7-TRACK MAG TAPE INPUT AND OUTPUT IS TO BE BINARY OR BCD. A VALUE OF ZERO SPECIFIES BINARY (3 TAPE FRAMES PER WORD, ALL 16-BITS OF EVERY WORD WITH TWO HIGH ORDER BITS OF ZERO) WHILE A NON-ZERO VALUE SPECIFIES BCD (2 TAPE FRAMES PER WORD, LEAST SIGNIFICANT 12-BITS OF EVERY WORD). THE DEFAULT VALUE OF >BCD IS ZERO AND AFTER EVERY INPUT OR OUTPUT THE VALUE IS RESET TO ZERO. SEE MTR, MTREAD, MTW AND MTWRITE.

&gt;R

(C) &lt;VALUE&gt; &gt;R

PUSH <VALUE> ONTO THE RETURN STACK. SEE I AND R>.

?

&lt;ADDRESS&gt; ?

PRINT THE VALUE CONTAINED AT <ADDRESS> ON THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL), FREE FORMAT, CONVERTED ACCORDING TO THE CURRENT NUMBER BASE. EQUIVALENT TO THE SEQUENCE "<ADDRESS> @ .".

?DEF

(OLD) ?DEF <NAME> <LOGICAL-VALUE>  
RENAMED FIND.

?DUP

&lt;VALUE&gt; ?DUP IF ... THEN

IF <VALUE> IS NON-ZERO (WHICH WILL BE INTERPRETED AS A LOGICAL VALUE OF TRUE) THEN <VALUE> IS DUPLICATED ON THE STACK, OTHERWISE <VALUE> IS NOT DUPLICATED. USE OF THIS WORD ALLOWS ONE TO OMIT THE "ELSE DROP" CLAUSE FORM THE IF STATEMENT, WHEN IT IS DESIRED TO EXECUTE THE IF STATEMENT ONLY IF <VALUE> IS NON-ZERO.

- ?EOF**                    **?EOF <LOGICAL-VALUE>**  
 TESTS THE MAG TAPE AND RETURNS A <LOGICAL-VALUE> OF TRUE IF THE TAPE IS CURRENTLY POSITIONED AT AN END-OF-FILE MARK.
- ?LEFT**                    **?LEFT <CELLS>**  
 CALCULATES THE NUMBER OF MEMORY CELLS LEFT IN THE MEMORY OVERLAY REGION.
- ?MTREADY**                **?MTREADY <LOGICAL-VALUE>**  
 TESTS THE MAG TAPE FOR READY AND ONLINE AND IF THE TAPE DRIVE IS NOT READY THE MESSAGE "TAPE NOT READY" WILL BE OUTPUT TO THE TERMINAL AND THE WORD WAITS FOR THE TAPE DRIVE TO BE PLACED ONLINE.
- ?ON**                    **?ON <LOGICAL-RESULT>**  
 PUSH A <LOGICAL-RESULT> OF TRUE ONTO THE STACK IF THE LAST TOGGLE OF ANY CAMAC DISPLAY PANEL PUSHBUTTON'S STATUS BIT TURNED THE STATUS BIT ON, OTHERWISE A <LOGICAL-RESULT> OF FALSE IS PUSHED ONTO THE STACK. SEE PBARRAY, PLARRAY, PLTOGGLE, PSTOGGLE, PTOGGLE, ON! AND OFF!.
- ?PB**                    **<PUSHBUTTON#> ?PB <LOGICAL-RESULT>**  
 PUSH A <LOGICAL-RESULT> OF TRUE ONTO THE STACK IF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON'S STATUS BIT IS SET, OTHERWISE A <LOGICAL-RESULT> OF FALSE IS PUSHED ONTO THE STACK. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PBARRAY, ON!, OFF!, PSTOGGLE AND PTOGGLE.
- ?Q**                    **?Q <LOGICAL-VALUE>**  
 A CAMAC WORD TO TEST THE Q-RESPONSE. THE <LOGICAL-VALUE> PUT ON THE STACK CORRESPONDS TO WHETHER THE Q-RESPONSE IS TRUE OR FALSE.
- ?TER**                    **?TER <CHAR-CODE>**  
 PUSHES ONTO THE STACK THE 7-BIT ASCII CHARACTER CODE OF THE LAST CHARACTER ENTERED AT THE TERMINAL, OR ZERO IF NO CHARACTER HAS BEEN ENTERED. SEE APPENDIX A FOR A LISTING OF THE ASCII CODES.
- @**                    **<ADDRESS> @ <VALUE>**  
 FETCH THE CONTENTS OF THE MEMORY LOCATION <ADDRESS> AND PUSH IT ONTO THE STACK.
- @I/O**  
 RESTORES THE SYSTEM FLAGS AND PARAMETERS THAT WERE SAVED BY !I/O, PRIOR TO PERFORMING I/O FROM AN INTERRUPT HANDLING WORD. @I/O INCLUDES THE EXECUTION OF FRESTORE. SEE !I/O.
- @STATE**                **@STATE <LOGICAL-VALUE>**  
 PUSHES ONTO THE STACK A LOGICAL-VALUE WHICH IS TRUE ONLY IF FORTH IS IN COMPILATION MODE. THIS WORD IS USEFUL ONLY IN COMPILER DIRECTIVE WORDS.

- A+** (A)  
A CONSTANT WHOSE VALUE SPECIFIES THE JUMP CONDITION FOR THE "A REGISTER  $\geq$  0" TEST. USUALLY FOLLOWED BY IF, END, JIF, JIFM, OR XIF,. REFER TO PAGE 20-18 OF THE VARIAN HANDBOOK. SEE NOT.
- A-** (A)  
A CONSTANT WHOSE VALUE SPECIFIES THE JUMP-CONDITION FOR THE "A REGISTER  $<$  0" TEST. USUALLY FOLLOWED BY IF, END, JIF, JIFM, OR XIF,. REFER TO PAGE 20-18 OF THE VARIAN HANDBOOK. SEE NOT.
- A-SAVE** A-SAVE <NAME>  
WRITES THE CURRENT OVERLAY TO DISC, STARTING AT THE BLOCK NUMBER CONTAINED IN THE VARIABLE O-BLK. O-BLK IS THEN UPDATED ACCORDINGLY. <NAME> IS ENTERED INTO THE DICTIONARY SUCH THAT SUBSEQUENT EXECUTION OF <NAME> WILL PUSH THE STARTING BLOCK NUMBER OF THE OVERLAY ONTO THE STACK (SEE O-LOAD). SEE O-SAVE.
- AO** (A)  
A CONSTANT WHOSE VALUE SPECIFIES THE JUMP-CONDITION FOR THE "A REGISTER = 0" TEST. USUALLY FOLLOWED BY IF, END, JIF, JIFM, OR XIF,. REFER TO PAGE 20-18 OF THE VARIAN HANDBOOK. SEE NOT.
- ABORT**  
ENTER THE ABORT SEQUENCE WHICH WILL CLEAR BOTH THE STACK AND THE RETURN STACK, PRINT THE ABORT MESSAGE "?S" AND RETURN CONTROL TO THE OPERATOR'S TERMINAL. SEE QUIT.
- ABORT** (A)  
A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE TO PRODUCE A "?Q" ABORT MESSAGE. THE NORMAL SEQUENCE IS "ABORT JMP,", HOWEVER, THE FOLLOWING SEQUENCES WILL PRODUCE THE SPECIFIED ABORT:
- |            |      |              |    |       |
|------------|------|--------------|----|-------|
| ABORT 1-   | JMP, | WILL PRODUCE | ?R | ABORT |
| ABORT 2 -  | JMP, | WILL PRODUCE | ?S | ABORT |
| ABORT 3 -  | JMP, | WILL PRODUCE | ?T | ABORT |
| ABORT 4 -  | JMP, | WILL PRODUCE | ?U | ABORT |
| ABORT 5 -  | JMP, | WILL PRODUCE | ?V | ABORT |
| ABORT 6 -  | JMP, | WILL PRODUCE | ?W | ABORT |
| ABORT 7 -  | JMP, | WILL PRODUCE | ?X | ABORT |
| ABORT 10 - | JMP, | WILL PRODUCE | ?Y | ABORT |
| ABORT 11 - | JMP, | WILL PRODUCE | ?Z | ABORT |
- ABS** <VALUE> ABS <RESULT>  
FORM THE ABSOLUTE VALUE OF <VALUE> AND LEAVE IT ON THE STACK.
- ACURSOR** ACURSOR <STATUS> <Y-POSN> <X-POSN>  
TURNS ON THE 4010 CROSS HAIR CURSORS AND WAITS FOR THE OPERATOR TO ENTER ANY CHARACTER. THREE SINGLE-WORD INTEGERS ARE RETURNED, THE X AND Y POSITIONS OF THE CURSORS AND THE 4010 STATUS WORD.
- ADD,** (A) <ADDRESS> ADD,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ADD INSTRUCTION (ADD MEMORY TO THE A REGISTER).

- ADM, (A) <ADDRESS> ADM,  
AN ASSEMBLER MACRO WHICH GENERATES A SEQUENCE OF MACHINE INSTRUCTIONS TO ADD THE CONTENTS OF THE A REGISTER TO THE CONTENTS OF THE SPECIFIED MEMORY LOCATION. THE SEQUENCE OF INSTRUCTIONS GENERATE IS ADD AND STA.
- ADOPT (OLD) ADOPT  
SAME AS , EXECPT THAT ADOPT IS A COMPILER DIRECTIVE.
- ADOX (C) ADOX  
ADOPT AN ANONYMOUS COLON DEFINITION OR CODE DEFINITION BY PLACING THE ADDRESS CONTAINED IN PSEUDO-VECTOR ENTRY X INTO THE DICTIONARY. SEE P-VX, :ORX AND ORCX.
- ANA, (A) <ADDRESS> ANA,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ANA INSTRUCTION (LOGICAL AND MEMORY WITH THE A REGISTER).
- AND <VALUE1> <VALUE2> AND <RESULT>  
CALCULATE THE BITWISE LOGICAL-AND OF <VALUE1> AND <VALUE2>, LEAVING THE RESULT ON THE STACK.
- ASHIFT <DW-VALUE> <SHIFT-COUNT> ASHIFT <DW-RESULT>  
ARITHMETIC SHIFT THE <DW-VALUE>, LEFT FOR A POSITIVE <SHIFT-COUNT> AND RIGHT FOR A NEGATIVE <SHIFT-COUNT>. THIS WORD MAY BE USED TO MULTIPLY AND DIVIDE A DOUBLE-WORD VALUE BY A POWER OF 2.
- ASL, (A) <SHIFT-COUNT> ASL,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ASLA INSTRUCTION (ARITHMETIC SHIFT LEFT THE A REGISTER).
- ASLB, (A) <SHIFT-COUNT> ASLB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ASLB INSTRUCTION (ARITHMETIC SHIFT LEFT THE B REGISTER).
- ASR, (A) <SHIFT-COUNT> ASR,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ASRA INSTRUCTION (ARITHMETIC SHIFT RIGHT THE A REGISTER).
- ASRB, (A) <SHIFT-COUNT> ASRB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ASRB INSTRUCTION (ARITHMETIC SHIFT RIGHT THE B REGISTER).
- ASK (OLD) ASK <VALUE>  
REQUEST THE INPUT OF A NUMBER FROM THE TERMINAL.
- ASSEMBLER (P)  
SWITCH THE CONTEXT POINTER SO THAT DICTIONARY SEARCHES WILL BEGIN IN THE ASSEMBLER VOCABULARY. THE ASSEMBLER VOCABULARY IS ALWAYS CHAINED TO THE CURRENT VOCABULARY.

- B!**                   <VALUE> <BYTE-ADDRESS> B!  
STORE THE LEAST SIGNIFICANT 8-BITS OF <VALUE> AT THE BYTE OF MEMORY POINTED TO BY <BYTE-ADDRESS>.
- B)**                   (A)  
SETS THE VARIABLE MODE TO 6, SPECIFYING INDEXING OFF THE B REGISTER FOR THE NEXT MEMORY REFERENCE INSTRUCTION.
- B.**                   <VALUE> B.  
BINARY OUTPUT. OUTPUT <VALUE> AS A BINARY (BASE 2) NUMBER, UNSIGNED AND PRECEDED BY A BLANK ON THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE FORMAT SPECIFICATIONS GIVEN BY THE VARIABLES FLD AND DPL ARE OBSERVED. BASE IS NOT CHANGED.
- B@**                   <BYTE-ADDRESS> B@ <RESULT>  
FETCH THE 8-BIT BYTE FROM THE BYTE OF MEMORY POINTED TO BY <BYTE-ADDRESS> AND LEAVE THIS BYTE ON THE STACK. THE HIGH ORDER 8 BITS OF <RESULT> WILL ALWAYS BE ZERO.
- BO**                   (A)  
A CONSTANT WHOSE VALUE SPECIFIES THE JUMP-CONDITION FOR THE "B REGISTER = 0" TEST. USUALLY FOLLOWED BY IF, END, JIF, JIFM, OR XIF,. REFER TO PAGE 20-18 OF THE VARIAN HANDBOOK. SEE NOT.
- BASE**  
A VARIABLE CONTAINING THE CURRENT NUMBER CONVERSION BASE. THE WORD DECIMAL SETS BASE TO 10, OCTAL SETS BASE TO 8, HEX SETS BASE TO 16, ETC.
- BEGIN**               (CO+,P) BEGIN ... <LOGICAL-VALUE> END  
                      BEGIN ... <LOGICAL-VALUE> WHILE ... REPEAT  
BEGIN MARKS THE START OF A SEQUENCE OF WORDS THAT ARE TO BE EXECUTED REPEATEDLY UNTIL A SPECIFIED CONDITION IS TRUE. IF THE BEGIN-END FORM IS USED, ALL THE WORDS BETWEEN THE BEGIN AND THE END ARE EXECUTED REPEATEDLY UNTIL THE <LOGICAL-VALUE> IS TRUE, AT WHICH POINT THE WORDS FOLLOWING THE END ARE EXECUTED. IF THE BEGIN-WHILE-REPEAT FORM IS USED THE WORDS BETWEEN THE BEGIN AND THE REPEAT ARE EXECUTED REPEATEDLY AS LONG AS THE <LOGICAL-VALUE> ENCOUNTERED BY WHILE IS TRUE. WHEN WHILE ENCOUNTERS A FALSE <LOGICAL-VALUE> THE LOOP IS EXITED IMMEDIATELY. THESE LOOPS MAY BE NESTED.
- BEGIN,**               (A)       BEGIN, ... <JUMP-CONDITION> END,  
BEGIN, MARKS THE START OF A SEQUENCE OF MACHINE INSTRUCTIONS THAT ARE TO BE EXECUTED REPEATEDLY UNTIL THE SPECIFIED <JUMP-CONDITION> IS TRUE. <JUMP-CONDITION> IS USUALLY SPECIFIED BY ONE OF THE WORDS A+, A-, AO, BO OR OV. BEGIN,-END, LOOPS MAY BE NESTED.
- BELL**  
ACTIVATE THE TERMINAL BELL OR NOISEMAKER, AS APPROPRIATE FOR THE TERMINAL DEVICE.

- BINARY** (A)  
A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE IN FORTH TO POP TWO WORDS OFF THE STACK AND THEN PUSH THE CONTENTS OF THE A REGISTER ONTO THE STACK. THE SEQUENCE "BINARY 1-" LEAVES THE ADDRESS OF THE INTERPRETER ROUTINE TO POP THREE WORDS OFF THE STACK AND THEN PUSH THE CONTENTS OF THE A REGISTER ONTO THE STACK. THE NORMAL SEQUENCE IS EITHER "BINARY JMP," OR "BINARY 1- JMP,".
- BINEX**  
A VARIABLE USED TO HOLD THE BINARY EXPONENT OF A FLOATING-POINT NUMBER BY THE FLOATING-POINT ROUTINES.
- BK**  
INITIATE THE BACKSPACING OF A SINGLE MAG TAPE RECORD AND RETURN IMMEDIATELY. SEE BKSP.
- BKSP**  
INITIATE THE BACKSPACING OF A SINGLE MAG TAPE RECORD AND WAIT FOR THE OPERATION TO COMPLETE. SEE BK.
- BL**  
A CONSTANT WHOSE VALUE IS 32 (40B), NAMELY AN ASCII SPACE.
- BLANK** (OLD)  
RENAMED SPACE.
- BLK**  
A VARIABLE CONTAINING THE NUMBER OF THE BLOCK BEING LISTED OR EDITED.
- BLOCK** <BLOCK#> BLOCK <ADDRESS>  
FETCH THE SPECIFIED BLOCK FROM DISC OR TAPE AND LEAVE IT IN ONE OF FORTH'S BLOCK BUFFERS. THE STARTING MEMORY ADDRESS OF THE BLOCK IS THEN RETURNED ON THE STACK. IF THE SPECIFIED BLOCK IS ALREADY IN MEMORY THEN IT NEED NOT BE READ IN FROM SECONDARY STORAGE. SEE HBLOCK.
- BLOCK-ASK**  
SETS A FLAG SO THAT THE NEXT USE OF ANY ASKING WORD (SASK, DASK OR FASK) WILL FETCH CHARACTERS FROM THE BLOCK BUFFER RATHER THAN THE TERMINAL. SEE TERMINAL-ASK.
- BLOCK-WORD** (OLD)  
RENAMED BLOCK-ASK.
- BLOCKPRINT** <START-BLOCK#> <END-BLOCK#> BLOCKPRINT  
OUTPUT THE SPECIFIED SEQUENCE OF BLOCKS TO THE LINE PRINTER. THE APPROPRIATE LINE PRINTER CODE MUST HAVE PREVIOUSLY BEEN LOADED (SEE UTIL AND PRINTERS).

**BNOT** (OLD)  
RENAMED COM.

**BOLDFACE** INCREASE THE SIZE OF THE CHARACTERS PRINTED BY THE LINE PRINTER. THIS WORD IS DEVICE DEPENDENT.

**BOXGRID** <X-PHYSICAL-ORIGIN> <X-LOGICAL-ORIGIN> <X-LOGICAL-MAX>  
<X-PHYSICAL-SIZE> <Y-PHYSICAL-ORIGIN> <Y-LOGICAL-ORIGIN>  
<Y-LOGICAL-MAX> <Y-PHYSICAL-SIZE> BOXGRID  
DEFINES A LOGICAL COORDINATE SYSTEM FOR THE 4010 BASED ON THE EIGHT FLOATING-POINT PARAMETERS. IN ADDITION, A BOX IS DRAWN TO SURROUND THE PLOT AND THE AXES ARE LABELLED. THE PHYSICAL-ORIGIN AND THE PHYSICAL-SIZE ARE GIVEN IN PHYSICAL COORDINATES (0.0 THROUGH 1023.0 FOR X; 0.0 THROUGH 780.0 FOR Y). THE LOGICAL-ORIGIN AND THE LOGICAL-MAX ALLOWS THE USER TO IMPLICITLY DEFINE A LOGICAL COORDINATE SYSTEM WHICH THEN MAPS INTO THE SPECIFIED PHYSICAL COORDINATE SYSTEM. SEE L PLOT AND P PLOT.

**BRACKET** BRACKET <NAME>  
SEARCHES THE DICTIONARY FOR <NAME> AND OUTPUTS THE DICTIONARY ADDRESS AND THE THREE CHARACTER/COUNT IDENTIFIERS OF THE FOUR WORDS THAT PRECEDE <NAME> IN THE DICTIONARY AND THE FOUR WORDS THAT FOLLOW <NAME> IN THE DICTIONARY.

**BT,** (A) <ADDRESS> <VALUE> BT,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN BT INSTRUCTION (BIT TEST).

**BUFFER** <BLOCK#> BUFFER <ADDRESS>  
OBTAIN A CORE BUFFER FOR THE SPECIFIED BLOCK AND LEAVE THE STARTING ADDRESS OF THE BUFFER ON THE STACK. THE BLOCK IS NOT READ FROM DISC AND IS AUTOMATICALLY MARKED AS UPDATED (I.E. - THE CONTENTS OF THE CORE BUFFER WILL BE WRITTEN ONTO DISC OR TAPE WHEN THE BUFFER SPACE IS NEEDED FOR ANOTHER BLOCK). USE THIS WORD RATHER THAN BLOCK WHEN AN ENTIRE BLOCK IS GOING TO BE RE-WRITTEN, TO REDUCE DISC ACCESSSES.

**BYTE** <ADDRESS> BYTE <BYTE-ADDRESS>  
CONVERT THE MEMORY <ADDRESS> TO A BYTE-ADDRESS AND LEAVE THE BYTE-ADDRESS ON THE STACK. THIS CONVERSION IS A LOGICAL ROTATE OF THE MEMORY ADDRESS ONE BIT LEFT. NOTE THAT THIS WORD IS EQUIVALENT TO A MULTIPLICATION BY 2 ONLY IF THE MULTIPLICAND IS POSITIVE. IF THE MULTIPLICAND IS NEGATIVE THEN A LOGICAL ROTATE LEFT IS NOT EQUIVALENT TO A MULTIPLICATION BY 2.



## C.DRG

A VARIABLE WHOSE VALUE SPECIFIES THE STARTING ADDRESS OF A BUFFER CONTAINING THE DOUBLE-WORD DATA POINTS FOR AN FFT. SEE DFOURTRAN, DINVTRAN AND LENGTH.

CASE (C2+,P) <VALUE1> <VALUE2> CASE ... THEN  
                   <VALUE1> <VALUE2> CASE ... ELSE ... THEN  
 IF <VALUE1> EQUALS <VALUE2>, DROP BOTH <VALUE1> AND <VALUE2>  
 AND EXECUTE THE WORDS DIRECTLY FOLLOWING CASE, UP TO THE NEXT  
 ELSE OR THEN. IF <VALUE1> DOES NOT EQUAL <VALUE2> THEN <VALUE2>  
 IS DROPPED BUT <VALUE1> IS LEFT ON THE STACK AND THE WORDS  
 FOLLOWING THE ELSE (OR THE THEN, IF NO ELSE IS PRESENT) ARE  
 EXECUTED. NOTE THAT EVERY CASE MUST HAVE A TERMINATING THEN.  
 CASE IS EQUIVALENT TO THE SEQUENCE "OVER = IF DROP". CASE IS  
 USED TO COMPARE <VALUE1> AGAINST A LIST OF POSSIBLE VALUES, FOR  
 EXAMPLE:  
 <VALUE1> <VALUE2> CASE (ACTION FOR <VALUE2>) ELSE  
           <VALUE3> CASE (ACTION FOR <VALUE3>) ELSE  
           <VALUE4> CASE (ACTION FOR <VALUE4>) ELSE  
                   ELSE (NOT EQUAL TO ANY OF THE ABOVE)  
                   THEN THEN THEN THEN

## CELL

<BYTE-ADDRESS> CELL <ADDRESS>  
 CONVERT THE <BYTE-ADDRESS> TO ITS MEMORY ADDRESS, LEAVING THE  
 MEMORY ADDRESS ON THE STACK. THIS CONVERSION IS A LOGICAL SHIFT  
 OF THE BYTE-ADDRESS ONE BIT RIGHT. NOTE THAT THIS WORD IS  
 EQUIVALENT TO A DIVISION BY 2 ONLY IF THE DIVIDEND IS POSITIVE.  
 IF THE DIVIDEND IS NEGATIVE THEN A LOGICAL SHIFT RIGHT IS NOT  
 EQUIVALENT TO A DIVISION BY 2.

## CHAIN

CHAIN <NAME>  
 CONNECT THE CURRENT VOCABULARY TO ALL DEFINITIONS THAT MIGHT BE  
 ENTERED INTO VOCABULARY <NAME> IN THE FUTURE. THE CURRENT  
 VOCABULARY MAY NOT BE FORTH OR ASSEMBLER. ANY GIVEN VOCABULARY  
 MAY ONLY BE CHAINED ONCE BUT MAY BE THE OBJECT OF ANY NUMBER OF  
 CHAININGS. FOR EXAMPLE, EVERY USER-DEFINED VOCABULARY MAY  
 INCLUDE THE SEQUENCE "CHAIN FORTH".

## CHAR

<MAX#CHARACTERS> CHAR <NAME>  
 DEFINE A CHARACTER STRING. ENOUGH ROOM IS ALLOCATED FOR  
 <MAX#CHARACTERS> AND THE DICTIONARY ENTRY IS IDENTIFIED BY  
 <NAME>. <MAX#CHARACTERS> IS SAVED IN THE DICTIONARY ENTRY FOR  
 <NAME> SO THAT ONE CAN ALWAYS DETERMINE THE MAXIMUM NUMBER OF  
 CHARACTERS THE STRING MAY HOLD, REGARDLESS OF THE NUMBER OF  
 CHARACTERS CURRENTLY CONTAINED IN THE STRING. SUBSEQUENT  
 EXECUTION OF <NAME> WILL PUSH ONTO THE STACK THE ADDRESS OF THE  
 FIRST TWO BYTES OF THE CHARACTER STRING (THE COUNT BYTE AND THE  
 FIRST CHARACTER), AS REQUIRED, FOR EXAMPLE, BY COUNT. SEE  
 CMOVE.

- CHCURSOR**                    **CHCURSOR** <Y-POSN> <X-POSN>  
 TURNS ON THE 4010 CROSS HAIR CURSORS AND WAITS FOR THE OPERATOR TO ENTER ANY CHARACTER. TWO SINGLE-WORD INTEGERS ARE RETURNED, THE X AND Y POSITIONS OF THE CURSORS.
- CHFETCH**                    **CHFETCH** <CHAR-CODE>  
 RETURNS THE CHARACTER WHICH IS STORED IN THE BYTE POINTED TO BY THE BYTE-ADDRESS IN IP. IP IS ALSO INCREMENTED.
- CIA,**                    (A)                    <DEVICE-CODE> CIA,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN CIA INSTRUCTION (CLEAR AND INPUT TO THE A REGISTER).
- CIB,**                    (A)                    <DEVICE-CODE> CIB,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN CIB INSTRUCTION (CLEAR AND INPUT TO THE B REGISTER).
- CMOVE**                    <SOURCE-ADDRESS> <DESTINATION-ADDRESS> CMOVE  
 MOVE A CHARACTER STRING FROM THE SPECIFIED SOURCE ADDRESS TO THE SPECIFIED DESTINATION ADDRESS. THE DESTINATION ADDRESS IS ASSUMED TO POINT TO THE DICTIONARY ENTRY FOR A WORD DEFINED AS A FORTH CHARACTER STRING (SEE CHAR). THE DICTIONARY ENTRY FOR THE DESTINATION FIELD WILL THEN SPECIFY THE MAXIMUM NUMBER OF CHARACTERS THAT THE STRING MAY HOLD AND ANY EXCESS CHARACTERS IN THE SOURCE FIELD WILL NOT BE MOVED. BOTH <SOURCE-ADDRESS> AND <DESTINATION-ADDRESS> MUST POINT TO THE FIRST TWO BYTES OF THEIR RESPECTIVE CHARACTER STRINGS (THE COUNT BYTE AND THE FIRST CHARACTER).
- CODE**                    **CODE** <NAME>  
 CREATE A DICTIONARY ENTRY DEFINING <NAME> AS EQUIVALENT TO THE SEQUENCE OF ASSEMBLER CODE THAT FOLLOWS <NAME>. THE CONTEXT VOCABULARY IS SET TO ASSEMBLER. IT IS VERY IMPORTANT TO REMEMBER THAT FORTH'S COMPILATION FLAG IS NOT SET WHILE ASSEMBLING MACHINE CODE INSTRUCTIONS, THAT IS, FORTH REMAINS IN EXECUTION MODE.
- COM**                    <VALUE> COM <RESULT>  
 FORM THE ONES-COMPLEMENT OF <VALUE> AND LEAVE IT ON THE STACK.
- COMP,**                    (A)  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN COMP INSTRUCTION (COMPLEMENT AND COMBINE REGISTERS).
- CONSTANT**                    <VALUE> CONSTANT <NAME>  
 DEFINE A WORD <NAME> WHICH WHEN EXECUTED WILL PUSH ITS SINGLE-WORD VALUE ONTO THE STACK. THE VALUE OF <NAME> IS INITIALIZED TO <VALUE>. THE <VALUE> OF THE CONSTANT MAY BE CHANGED BY THE SEQUENCE "<NEW-VALUE> ' <NAME> !".
- CONTEXT**  
 A VARIABLE CONTAINING A POINTER TO THE VOCABULARY IN WHICH DICTIONARY SEARCHES ARE TO BEGIN. SEE CURRENT.

- CONTINUED (E)      <BLOCK#> CONTINUED  
 CONTINUE INTERPRETATION WITH THE SPECIFIED BLOCK. THE SEQUENCE "1 +BLOCK CONTINUED" CONTINUES INTERPRETATION WITH THE NEXT BLOCK.
- COUNT      <ADDRESS> COUNT <COUNT>  
 <ADDRESS> POINTS TO THE FIRST TWO BYTES OF A FORTH CHARACTER STRING AND COUNT WILL RETURN THE NUMBER OF CHARACTERS IN THE STRING ON TOP OF THE STACK AND THE BYTE-ADDRESS OF THE STRING WILL BE STORED IN IP. IT IS ASSUMED THAT THE FIRST BYTE OF THE STRING IS THE CHARACTER COUNT AND THE ACTUAL STRING STARTS WITH THE SECOND BYTE. THIS WORD IS USUALLY FOLLOWED BY EITHER WRITE OR TYPE.
- CPA,      (A)  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN CPA INSTRUCTION (COMPLEMENT THE A REGISTER).
- CPU      (A)      <VALUE> CPU <NAME>  
 DEFINE <NAME> AS A SINGLE-WORD MACHINE INSTRUCTION WHOSE MACHINE CODE REPRESENTATION IS <VALUE>. WHEN <NAME> IS EXECUTED IT WILL REQUIRE NO PARAMETERS ON THE STACK AND <VALUE> WILL BE STORED IN THE NEXT AVAILABLE DICTIONARY LOCATION. SEE DL, IO AND M/CPU.
- CR  
 OUTPUTS A CARRIAGE-RETURN, LINE-FEED TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL).
- CRATE  
 A VARIABLE USED BY THE CAMAC WORDS TO CONTAIN THE CRATE NUMBER FOR CAMAC DEFINITIONS WHICH ARE BEING COMPILED INTO THE DICTIONARY. THE DEFAULT VALUE OF THIS VARIABLE IS 1.
- CUR      (T)  
 A VARIABLE CONTAINING THE PHYSICAL RECORD NUMBER BEFORE WHICH THE MAG TAPE IS CURRENTLY POSITIONED. REWIND SETS CUR TO ZERO.
- CURRENT  
 A VARIABLE CONTAINING A POINTER TO THE VOCABULARY INTO WHICH NEW WORDS ARE TO BE ENTERED. THE SEQUENCE "CURRENT @ @" LEAVES THE LINK ADDRESS OF THE NEXT ENTRY TO BE DEFINED.
- CURSE      CURSE <CHAR-CODE> <Y-POSN> <X-POSN>  
 TURNS ON THE 4010 CROSS HAIR CURSORS AND WAITS FOR THE OPERATOR TO ENTER ANY CHARACTER. THE CURSOR POSITION IS SAVED IN /CURSE. <X-POSN> AND <Y-POSN> ARE FLOATING-POINT VALUES SPECIFYING THE PHYSICAL POSITION OF THE CURSOR CROSS HAIR AND <CHAR-CODE> IS THE ASCII CHARACTER CODE FOR THE CHARACTER THAT THE OPERATOR ENTERED. THE TERMINAL IS PUT BACK INTO ALPHA MODE WITH THE CURSOR AT THE SAVED POSITION.

- D** (OLD)  
A VARIABLE SPECIFYING EITHER THE NUMBER OF DIGITS TO THE RIGHT OF THE RADIX POINT IN THE LAST NUMBER INPUT (A NEGATIVE VALUE IF THERE WAS NO RADIX POINT ENTERED) OR THE NUMBER OF DIGITS TO FOLLOW THE RADIX POINT IN THE NEXT NUMBER TO BE OUTPUT. REPLACED BY #D FOR INPUT AND DPL FOR OUTPUT.
- D!**                   <DW-VALUE> <ADDRESS> D!  
STORE <DW-VALUE> STARTING AT THE SPECIFIED MEMORY ADDRESS.
- D\***                   <DW-FRACTION1> <DW-FRACTION2> D\* <DW-FRACTION-RESULT>  
MULTIPLY THE TWO DOUBLE-WORD FRACTIONS LEAVING THE RESULT, A DOUBLE-WORD FRACTION ON THE STACK. NOTE THAT THE MULTIPLICATION OF TWO FRACTIONS WILL ALWAYS GENERATE A FRACTIONAL RESULT. NOTE THAT THIS WORD DOES NOT MULTIPLY DOUBLE-WORD INTEGERS BUT MULTIPLIES DOUBLE-WORD FRACTIONS.
- D\*/**                  <DW-VALUE> <VALUE1> <VALUE2> D\*/ <DW-RESULT>  
MULTIPLY <DW-VALUE> BY <VALUE1> AND THEN DIVIDE THE RESULT BY <VALUE2>, LEAVING THE RESULT, A DOUBLE-WORD INTEGER, ON THE STACK.
- D+**                   <DW-VALUE1> <DW-VALUE2> D+ <DW-RESULT>  
DOUBLE-WORD INTEGER ADDITION, LEAVING THE RESULT ON THE STACK.
- D-**                   <DW-VALUE1> <DW-VALUE2> D- <DW-RESULT>  
DOUBLE-WORD INTEGER SUBTRACTION, LEAVING THE RESULT, <DW-VALUE1> - <DW-VALUE2>, ON THE STACK.
- D-H**                (T)  
LOADS THE DISC HANDLERS. SEE UTIL.
- D.**                   <DW-VALUE> D.  
DOUBLE-WORD INTEGER OUTPUT. OUTPUT <DW-VALUE> TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE FIELD WIDTH IS SPECIFIED BY FLD AND THE NUMBER OF PLACES TO THE RIGHT OF THE RADIX POINT ARE SPECIFIED BY DPL.
- D/**                   <DW-FRACTION1> <DW-FRACTION2> D/ <DW-FRACTION-RESULT>  
DOUBLE-WORD FRACTIONAL DIVIDE, LEAVING THE RESULT, <DW-FRACTION1> / <DW-FRACTION2>, ON THE STACK. NOTE THAT THIS WORD DOES NOT DIVIDE DOUBLE-WORD INTEGERS BUT DIVIDES DOUBLE-WORD FRACTIONS.

- DO** (OLD)  
RENAMED DO,.
- DO,**  
A 2CONSTANT WHOSE VALUE IS THE DOUBLE-WORD INTEGER 0.
- DO<** <DW-VALUE> DO< <LOGICAL-VALUE>  
**DO=** <DW-VALUE> DO= <LOGICAL-VALUE>  
 COMPARE <DW-VALUE> AGAINST ZERO AND LEAVE A <LOGICAL-VALUE> OF TRUE ON THE STACK IF THE INDICATED RELATION IS TRUE, OTHERWISE LEAVE A <LOGICAL-VALUE> OF FALSE ON THE STACK.
- D1** (OLD)  
RENAMED D1,.
- D1,**  
A 2CONSTANT WHOSE VALUE IS THE DOUBLE-WORD FRACTION 1. THIS VALUE IS NOT THE DOUBLE-WORD INTEGER 1.
- D2T** <BLOCK#> D2T  
TRANSFER THE SPECIFIED BLOCK FROM DISK TO TAPE. SEE D-H AND T-H.
- D<** <DW-VALUE1> <DW-VALUE2> D< <LOGICAL-VALUE>  
**D=** <DW-VALUE1> <DW-VALUE2> D= <LOGICAL-VALUE>  
**D>** <DW-VALUE1> <DW-VALUE2> D> <LOGICAL-VALUE>  
 COMPARE <DW-VALUE1> AND <DW-VALUE2> AND LEAVE A <LOGICAL-VALUE> OF TRUE ON THE STACK IF THE INDICATED RELATION IS TRUE, OTHERWISE LEAVE A <LOGICAL-VALUE> OF FALSE ON THE STACK.
- D@** <ADDRESS> D@ <DW-VALUE>  
FETCH THE DOUBLE-WORD VALUE STARTING AT MEMORY LOCATION <ADDRESS> AND PUSH IT ONTO THE STACK.
- DABS** <DW-VALUE> DABS <DW-RESULT>  
FORM THE ABSOLUTE VALUE OF <DW-VALUE> AND LEAVE IT ON THE STACK.
- DAR,** (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN DAR INSTRUCTION (DECREMENT THE A REGISTER).
- DASK** DASK <DW-VALUE>  
REQUEST THE INPUT OF A DOUBLE-WORD VALUE FROM THE TERMINAL.

DBR, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN DBR INSTRUCTION  
(DECREMENT THE B REGISTER).

DCONSTANT (OLD)  
RENAMED 2CONSTANT.

DECIMAL  
SETS THE NUMERIC CONVERSION BASE TO DECIMAL MODE, THAT IS, SET  
THE VARIABLE BASE TO 10. SEE OCTAL AND HEX.

DECR, (A) <VALUE> DECR,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN DECR INSTRUCTION  
(DECREMENT AND COMBINE REGISTERS).

DEFINITIONS <NAME> DEFINITIONS  
SET THE CURRENT VOCABULARY (INTO WHICH NEW DEFINITIONS ARE  
BEING ENTERED) TO THE VOCABULARY <NAME>. <NAME> NEED NOT BE  
SPECIFIED EXPLICITLY, IN WHICH CASE <NAME> IS ASSUMED TO BE THE  
CONTEXT VOCABULARY.

DELIMITER (OLD)  
A VARIABLE SPECIFYING THE CHARACTER THAT TERMINATES A WORD.

DFIX <FP-VALUE> DFIX <DW-RESULT>  
TRUNCATE THE FLOATING-POINT VALUE TO A DOUBLE-WORD INTEGER  
VALUE. IF ONE WANTS TO ROUND THE FLOATING-POINT VALUE PRIOR TO  
TRUNCATION, THE FLOATING-POINT VALUE 0.5 SHOULD BE ADDED TO  
<FP-VALUE> PRIOR TO EXECUTING DFIX. SEE SFIX.

DFLOAT <DW-VALUE> DFLOAT <FP-RESULT>  
CONVERT THE DOUBLE-WORD INTEGER TO A FLOATING-POINT VALUE.

DFOURTRAN  
PERFORM A RADIX 2 FAST FOURIER TRANSFORM USING THE COOLEY-TUKEY  
ALGORITHM. THE STARTING BUFFER ADDRESS OF THE INPUT DATA MUST  
BE STORED IN THE VARIABLE C.ORG AND THE NUMBER OF DATA POINTS  
(A POWER OF 2) MUST HAVE BEEN SPECIFIED BY EXECUTING THE WORD  
LENGTH. THE INPUT DATA IS A VECTOR OF REAL, DOUBLE-WORD  
INTEGERS AND THE RESULT (WHICH OVERWRITES THE INPUT DATA) IS A  
VECTOR OF COMPLEX, DOUBLE-WORD INTEGERS. SINCE THE FOURIER  
TRANSFORM OF N REAL POINTS IS HERMITIAN (REAL PART EVEN AND  
IMAGINARY PART ODD) ONLY (N/2)+1 COMPLEX POINTS ARE RETURNED.  
THE IMAGINARY PART OF THE FIRST AND LAST DATA POINT WILL ALWAYS  
BE ZERO SINCE THE IMAGINARY PART IS AN ODD FUNCTION. THE BUFFER  
FOR AN N-POINT FFT MUST BE OF LENGTH (N\*2)+4 WORDS (THE  
ADDITIONAL 4 WORDS ARE REQUIRED FOR THE LAST OF THE (N/2)+1  
COMPLEX POINTS RETURNED). SEE DINVTRAN.

## DHALF

A 2CONSTANT WHOSE VALUE IS 0.5 WHEN CONSIDERED AS A DOUBLE-WORD FRACTION.

## DHSIN

<DW-FRACTION> DHSIN <DW-FRACTION-RESULT>

REPLACE THE <DW-FRACTION>, GIVEN IN HALF CIRCLES, WITH ITS SINE (IN RADIANS) ALSO A DOUBLE-WORD FRACTION. THIS WORD IS THE BUILDING BLOCK FOR THE FLOATING-POINT TRIGONOMETRIC FUNCTIONS.

## DIGILIGHTS

<VALUE> <START#> <END#> DIGILIGHTS

WRITE <VALUE> TO THE SPECIFIED DIGILIGHTS ON THE CAMAC DISPLAY PANEL.

## DIGISWITCHES

<START#> <END#> DIGISWITCHES <RESULT>

READS IN A GROUP OF DIGISWITCHES FROM THE CAMAC DISPLAY PANEL. <START#> AND <END#> SPECIFY THE STARTING AND ENDING DIGISWITCH NUMBERS AND IF 4 OR FEWER DIGISWITCHES ARE SPECIFIED THE <RESULT> WILL BE A SINGLE-WORD INTEGER WHILE IF 5 OR MORE DIGISWITCHES ARE SPECIFIED, <RESULT> WILL BE A DOUBLE-WORD INTEGER.

## DINVTRAN

PERFORM A RADIX 2 INVERSE FAST FOURIER TRANSFORM USING THE COOLEY-TUKEY ALGORITHM. THIS OPERATION IS THE INVERSE OF THE FFT PERFORMED BY DFOURTRAN, THAT IS, THE INPUT VECTOR CONTAINS (N/2)+1 COMPLEX, DOUBLE-WORD INTEGERS AND THE RESULT IS N REAL, DOUBLE-WORD INTEGERS.

## DISCARD

A NULL DEFINITION INTENDED FOR USE AS A STANDARD REMEMBER WORD. THIS NULL DEFINITION GUARANTEES THAT THE WORD DISCARD WILL ALWAYS BE FOUND. SEE FORGET AND REMEMBER.

## DISK

(T)

SETS THE DISC AS THE PRIMARY MASS STORAGE DEVICE. SEE D-H.

## DISK-TO-TAPE

<START-BLOCK#> <END-BLOCK#> DISK-TO-TAPE

TRANSFERS ALL NON-ZERO BLOCKS IN THE SPECIFIED RANGE FROM DISK TO TAPE. SEE D-H.

## DISKO

ZERO BLOCKS 1 THROUGH 511 ON THE DISC. EQUIVALENT TO THE SEQUENCE "1 511 ZERODISK". SEE UTIL.

- DIV,** (A) <ADDRESS> DIV,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN DIV INSTRUCTION (DIVIDE THE A AND B REGISTERS BY MEMORY).
- DL** (A) <VALUE> DL <NAME>  
DEFINE <NAME> AS A DOUBLE-WORD MACHINE INSTRUCTION WHOSE BASIC MACHINE CODE REPRESENTATION IS <VALUE>. WHEN <NAME> IS EXECUTED THE TOP NUMBER ON THE STACK IS INCLUSIVELY OR-ED WITH BOTH <VALUE> AND THE CURRENT VALUE OF MODE. THIS 16-BIT VALUE IS STORED IN THE NEXT AVAILABLE DICTIONARY LOCATION AS THE FIRST WORD OF THE INSTRUCTION. THE SECOND NUMBER ON THE STACK IS THEN STORED IN THE NEXT AVAILABLE DICTIONARY LOCATION AS THE SECOND WORD OF THE INSTRUCTION. SEE CPU, I/O AND M/CPU.
- DLIST** (E) <ADDRESS> DLIST  
LIST ALL WORDS IN THE DICTIONARY, STARTING WITH THE DICTIONARY ENTRY POINTED TO BY <ADDRESS>. WHEN ONE WISHES TO LIST ALL ENTRIES BEFORE A SPECIFIC ENTRY, THE USUAL SEQUENCE IS "'<NAME> DLIST".
- DMINUS** <DW-VALUE> DMINUS <DW-RESULT>  
NEGATE <DW-VALUE> BY FORMING ITS TWOS-COMPLEMENT.
- DO** (C) <ENDING-INCREMENT> <STARTING-INCREMENT> DO ...  
BEGIN A DO LOOP WHICH MUST THEN BE TERMINATED BY EITHER LOOP OR +LOOP. THE LOOP INDEX BEGINS AT <STARTING-INCREMENT> AND IS THEN EITHER INCREMENTED OR DECREMENTED EACH TIME THROUGH THE LOOP BY THE WORDS LOOP OR +LOOP.
- DOUBLE** (OLD)  
RENAMED 2VARIABLE.
- DP**  
A VARIABLE WHOSE VALUE IS THE ADDRESS OF THE NEXT AVAILABLE WORD IN THE DICTIONARY. SEE HERE AND DP+!.
- DP\***  
THE ADDRESS OF A SUBROUTINE TO PERFORM DOUBLE-WORD FRACTIONAL MULTIPLICATION. THIS SUBROUTINE IS CALLED BY THE SEQUENCE "DP\* 6 JSR,".
- DP+** DP+  
THE ADDRESS OF A SUBROUTINE TO PERFORM DOUBLE-WORD INTEGER ADDITION. THIS SUBROUTINE IS CALLED BY THE SEQUENCE "DP+ 6 JSR,".
- DP+!** <VALUE> DP+!  
ADD THE SIGNED <VALUE> TO THE DICTIONARY POINTER, DP. SINCE THE DICTIONARY POINTER MAY BE AN INTERNAL REGISTER RATHER THAN A VARIABLE, IT SHOULD ONLY BE ACCESSED THROUGH THE WORDS HERE AND DP+!.



## DPL

A VARIABLE WHICH ONE SETS TO THE NUMBER OF DIGITS THAT THEY WISH TO FOLLOW THE RADIX POINT IN A NUMBER TO BE OUTPUT. IF DPL IS SET TO A NEGATIVE VALUE, THE RADIX POINT WILL NOT BE PRINTED. SEE FLD AND W.D.

## DPLT

<X-POSN> <Y-POSN> DPLT

DRAW A VECTOR ON THE 4010 FROM THE CURRENT POSITION TO THE SPECIFIED OFFSET FROM THE CURRENT POSITION AS GIVEN BY <X-POSN> AND <Y-POSN>. THE PARAMETERS TO THIS WORD ARE OFFSETS FROM THE CURRENT POSITION AND NOT PHYSICAL COORDINATES. <X-POSN> AND <Y-POSN> ARE SINGLE-WORD INTEGERS IN THE RANGE 0-1023 AND 0-780.

## DPOLYVAL

<#TERMS> DPOLYVAL <NAME>

DEFINE <NAME> AS A WORD WHICH WHEN EXECUTED, WILL EVALUATE A POLYNOMIAL EXPRESSION. THE POLYNOMIAL MUST BE OF THE FORM  $Y = AO + X(A1 + X(A2 + \dots))$  AND IS EVALUATED ACCORDING TO HORNER'S RULE (FROM THE INNERMOST LEVEL OUTWARDS). FOLLOWING <NAME> THE COEFFICIENTS AO, A1, A2, ... MUST APPEAR AS DOUBLE-WORD FRACTIONS (THE WORD FD, IS HANDY FOR THIS) IN REVERSE ORDER (THAT IS, THE LAST TERM FIRST AND AO LAST). SINCE THE HIGHER COEFFICIENTS ARE ALMOST ALWAYS THE SMALLEST IN VALUE, THE SMALLER TERMS ARE ADDED FIRST IN ORDER TO MINIMIZE TRUNCATION ERRORS.

## DPREC

(OLD)

SETS AN INTERNAL FLAG TO INDICATE THAT NUMBERS CONTAINING A PERIOD ARE TO BE INTERPRETED AS DOUBLE-WORD INTEGERS, NOT AS FLOATING-POINT NUMBERS. SEE FLOATING.

## DROP

<VALUE> DROP

DROP THE TOP <VALUE> FROM THE STACK.

## DUMP

<STARTING-ADDRESS> <#CELLS> DUMP

DUMP THE CONTENTS OF <#CELLS> OF MEMORY, STARTING WITH <STARTING-ADDRESS> ONTO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). BOTH THE ADDRESS AND THE CONTENTS OF EACH WORD ARE PRINTED USING THE CURRENT NUMBER BASE.

## DUMP

(OLD) <STARTING-ADDRESS> DUMP

DUMP THE CONTENTS OF MEMORY, STARTING WITH <STARTING-ADDRESS>, ONTO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE OUTPUT IS TERMINATED BY PRESSING ANY TERMINAL KEY.

## DUP

<VALUE> DUP <VALUE> <VALUE>

DUPLICATE THE TOP <VALUE> ON THE STACK.

## DXR,

(A)

THE ASSEMBLER MNEMONIC FOR THE VARIAN DXR INSTRUCTION (DECREMENT THE X REGISTER). NOTE THAT THE X REGISTER IS THE STACK POINTER IN VARIAN FORTH, THEREFORE THIS INSTRUCTION ALLOCATES ONE MORE WORD OF THE STACK.

- E (OLD) <FP-NUMBER> E <VALUE> <FP-RESULT>  
THE GIVEN <FP-NUMBER> IS SCALED BY 10 \*\* <VALUE>.
- E) (A)  
SETS THE VARIABLE MODE TO 6, SPECIFYING INDEXING OFF THE B REGISTER FOR THE NEXT MEMORY REFERENCE INSTRUCTION. SINCE THE ADDRESS OF THE DICTIONARY ENTRY BEING EXECUTED IS CONTAINED IN THE B REGISTER THIS MNEMONIC IS MEANT TO INDICATE INDEXING OFF THE CURRENT DICTIONARY ENTRY.
- E. <FP-VALUE> E.  
OUTPUT THE FLOATING-POINT VALUE IN E-FORMAT, THAT IS, AS A FRACTION RAISED TO A POWER OF 10, TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE OUTPUT FORMAT IS SPECIFIED BY THE WORD W.D. THE FIELD WIDTH MUST INCLUDE THE 5 SPACES REQUIRED BY THE EXPONENT.
- E? <ADDRESS> E?  
EQUIVALENT TO THE SEQUENCE "F@ E.".
- EDIT <BLOCK#> EDIT  
EDIT THE SPECIFIED BLOCK. IF THE EDITOR IS NOT ALREADY LOADED INTO THE DICTIONARY IT WILL BE LOADED. THE BLOCK BEING EDITED IS FIRST LISTED.
- EDITOR (P)  
EDITOR IS THE NAME OF THE EDITOR VOCABULARY SO THAT IF THE EDITOR VOCABULARY HAS BEEN LOADED INTO THE DICTIONARY ONE MAY USE THE WORD EDITOR TO RE-INVOKE THE EDITOR VOCABULARY (FOLLOWING THE SWITCH TO SOME OTHER VOCABULARY).
- EJECT  
HAVE THE LINE PRINTER PAGE EJECT TO THE TOP OF THE NEXT PAGE.
- ELSE (C2,P)  
THE WORD ELSE IS USED IN AN IF-THEN CLAUSE TO SPECIFY WHERE THE IF BRANCH IS TO GO ON A FALSE <LOGICAL-VALUE>. THE ELSE CLAUSE IS OPTIONAL AND MAY BE OMITTED.
- ELSE, (A)  
THIS WORD IS USED IN AN IF,-THEN, CLAUSE TO SPECIFY WHERE TO GO ON A FALSE <JUMP-CONDITION>. THE ELSE, CLAUSE IS OPTIONAL AND MAY BE OMITTED.

- END (C2-,P) <LOGICAL-VALUE> END  
THIS WORD MARKS THE END OF A BEGIN-END LOOP. IF THE <LOGICAL-VALUE> IS TRUE THE LOOP IS TERMINATED, OTHERWISE CONTROL RETURNS TO THE FIRST WORD FOLLOWING THE CORRESPONDING BEGIN.
- END, (A) <JUMP-CONDITION> END,  
THIS WORD MARKS THE END OF A BEGIN,-END, LOOP. IF THE SPECIFIED <JUMP-CONDITION> IS TRUE THE LOOP IS TERMINATED, OTHERWISE A JUMP IS MADE BACK TO THE FIRST INSTRUCTION FOLLOWING THE BEGIN,.
- ENDFILE  
WRITE AN END-OF-FILE ON THE MAG TAPE AND WAIT FOR THE OPERATION TO COMPLETE. SEE WF.
- ERA, (A) <ADDRESS> ERA,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ERA INSTRUCTION (EXCLUSIVE-OR MEMORY WITH THE A REGISTER).
- ERASE-CORE  
MARK ALL BLOCK BUFFERS AS EMPTY. UPDATED BLOCKS ARE NOT FLUSHED. THE CONTENTS OF THE BUFFERS ARE SUBSEQUENTLY UNDEFINED.
- EXC, (A) <FUNCTION-DEVICE> EXC,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN EXC INSTRUCTION (EXTERNAL CONTROL TO A DEVICE).
- EXECUTE <ADDRESS> EXECUTE  
EXECUTE THE WORD SPECIFIED BY <ADDRESS>. <ADDRESS> MUST BE THE COMPILATION ADDRESS OF A DICTIONARY ENTRY, AS RETURNED BY THE WORD FIND. THE NORMAL SEQUENCE IS THEN "FIND <NAME> EXECUTE".
- EXIT (C)  
SKIP ONE LEVEL BACK, TO WHOEVER EXECUTED THE WORD IN WHICH EXIT EXISTS. THIS WORD PROVIDES A METHOD OF EXITING A WORD WITHOUT GOING ALL THE WAY THROUGH THE WORD TO THE SEMI-COLON.
- EXIT (OLD)  
RENAMED LEAVE.

- F (OLD)  
RENAMED FLD.
- F! <FP-VALUE> <ADDRESS> F!  
STORE <FP-VALUE> STARTING AT MEMORY LOCATION <ADDRESS>.
- F\* <FP-VALUE1> <FP-VALUE2> F\* <FP-RESULT>  
FLOATING-POINT MULTIPLICATION, LEAVING THE RESULT ON THE STACK.
- F+ <FP-VALUE1> <FP-VALUE2> F+ <FP-RESULT>  
FLOATING-POINT ADDITION, LEAVING THE RESULT ON THE STACK.
- F+! <FP-VALUE> <ADDRESS> F+!  
ADD <FP-VALUE> TO THE FLOATING-POINT VALUE STARTING AT MEMORY LOCATION <ADDRESS>. <FP-VALUE> MAY BE A POSITIVE OR A NEGATIVE VALUE. EQUIVALENT TO THE SEQUENCE "<ADDRESS> F@ <FP-VALUE> F+ <ADDRESS> F!".
- F- <FP-VALUE1> <FP-VALUE2> F- <FP-RESULT>  
FLOATING-POINT SUBTRACTION, LEAVING THE RESULT, <FP-VALUE1> - <FP-VALUE2>, ON THE STACK.
- F. <FP-VALUE> F.  
FLOATING-POINT OUTPUT. OUTPUT THE FLOATING-POINT VALUE TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE OUTPUT FORMAT MAY BE SPECIFIED BY THE WORD W.D.
- F/ <FP-VALUE1> <FP-VALUE2> F/ <FP-RESULT>  
FLOATING-POINT DIVIDE, LEAVING THE RESULT, <FP-VALUE1> / <FP-VALUE2>, ON THE STACK.
- F/620  
A CONSTANT DENOTING EITHER THE CPU TYPE OR THE INSTALLATION:  
     -1   = SOLAR 620/F WITH TELETYPE AND TEKTRONIX 611  
     0    = 620/L  
     1    = 620/F  
     2    = V74
- FO (OLD)  
RENAMED FO..
- FO.  
AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NUMBER 0.0.
- FO< <FP-VALUE> FO< <LOGICAL-VALUE>  
FO= <FP-VALUE> FO= <LOGICAL-VALUE>  
COMPARE <FP-VALUE> AGAINST 0.0 AND LEAVE A <LOGICAL-VALUE> OF TRUE IF THE INDICATED RELATION IS TRUE, OTHERWISE LEAVE A <LOGICAL-VALUE> OF FALSE ON THE STACK.

F10 (OLD)  
RENAMED F10..

F10.  
AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NUMBER 10.0.

F180.  
AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NUMBER 180.0.

F2LOG <FP-VALUE> F2LOG <FP-RESULT>  
COMPUTE THE LOGARITHM, BASE 2 OF <FP-VALUE> AND LEAVE THE  
RESULT ON THE STACK.

F2XP <FP-VALUE> F2XP <FP-RESULT>  
COMPUTE 2.0 \*\* <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.

F90.  
AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NUMBER 90.0.

F< <FP-VALUE1> <FP-VALUE2> F< <LOGICAL-VALUE>  
F= <FP-VALUE1> <FP-VALUE2> F= <LOGICAL-VALUE>  
F> <FP-VALUE1> <FP-VALUE2> F> <LOGICAL-VALUE>  
COMPARE <FP-VALUE1> AND <FP-VALUE2> AND LEAVE A <LOGICAL-VALUE>  
OF TRUE ON THE STACK IF THE INDICATED RELATION IS TRUE,  
OTHERWISE LEAVE A <LOGICAL-VALUE> OF FALSE ON THE STACK.

F? <ADDRESS> F?  
EQUIVALENT TO THE SEQUENCE "F@ F.".

FABS <FP-VALUE> FABS <FP-RESULT>  
REPLACE <FP-VALUE> BY ITS ABSOLUTE VALUE.

FASK FASK <FP-VALUE>  
REQUEST THE INPUT OF A FLOATING-POINT NUMBER FROM THE TERMINAL.

FCONSTANT <FP-VALUE> FCONSTANT <NAME>  
DEFINE THE WORD <NAME> WHICH WHEN EXECUTED WILL PUSH ITS  
FLOATING-POINT VALUE ONTO THE STACK. THE VALUE OF <NAME> IS  
INITIALIZED TO <FP-VALUE>. THE VALUE OF THIS CONSTANT MAY BE  
CHANGED BY EXECUTING THE SEQUENCE "<FP-VALUE> ' <NAME> F!".

FD, <FP-VALUE> FD,  
CONVERTS <FP-VALUE> TO A DOUBLE-WORD FRACTION AND PLACES THE  
FRACTION IN THE NEXT TWO DICTIONARY LOCATIONS.

- FDATN**                   <FP-VALUE1> <FP-VALUE2> FDATN <FP-RESULT>  
 COMPUTE THE ARCTANGENT (IN DEGREES) OF <FP-VALUE1> /  
 <FP-VALUE2>, LEAVING THE RESULT ON THE STACK. THE RESULT WILL  
 BE IN THE RANGE 0.0 THROUGH 359.999.
- FDCOS**                   <FP-VALUE> FDCOS <FP-RESULT>  
 COMPUTE THE COSINE OF <FP-VALUE> (IN DEGREES) AND LEAVE THE  
 RESULT ON THE STACK.
- FDSIN**                   <FP-VALUE> FDSIN <FP-RESULT>  
 COMPUTE THE SINE OF <FP-VALUE> (IN DEGREES) AND LEAVE THE  
 RESULT ON THE STACK.
- FDTAN**                   <FP-VALUE> FDTAN <FP-RESULT>  
 COMPUTE THE TANGENT OF <FP-VALUE> (IN DEGREES) AND LEAVE THE  
 RESULT ON THE STACK.
- FEXP**                   <FP-VALUE> FEXP <FP-RESULT>  
 COMPUTE  $E^{**}$  <FP-VALUE> (WHERE E IS THE BASE OF THE NATURAL  
 LOGARITHMS, 2.71828...) AND LEAVE THE RESULT ON THE STACK.
- FEXP10**                  <FP-VALUE> FEXP10 <FP-RESULT>  
 COMPUTE  $10.0^{**}$  <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.
- FHALF**  
 AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NUMBER 0.5.
- FIND**                   FIND <NAME> <RESULT>  
 IF A DICTIONARY ENTRY FOR <NAME> IS FOUND THEN FIND RETURNS THE  
 COMPILATION ADDRESS OF <NAME> (THE ADDRESS THAT WOULD NORMALLY  
 BE COMPILED WHEN <NAME> IS ENCOUNTERED IN A COLON-DEFINITION).  
 IF THE DICTIONARY ENTRY IS NOT FOUND THEN FIND RETURNS A VALUE  
 OF ZERO ON THE STACK.
- FIX**                   <STARTING-BLOCK#> <ENDING-BLOCK#> FIX  
 WORD REPLACEMENT. THE WORDS TO BE REPLACED MUST HAVE PREVIOUSLY  
 BEEN SPECIFIED USING THE WORDS REPLACE AND \$REPLACE. THE WORD  
 FIX THEN GOES THROUGH THE DESIGNATED BLOCKS AND REPLACES ALL  
 OCCURENCES OF THE WORDS THAT WERE SPECIFIED BY REPLACE AND  
 \$REPLACE WITH THEIR NEW REPRESENTATIONS. A LISTING OF ALL  
 CHANGES IS OUTPUT TO THE CURRENT OUTPUT DEVICE. BEWARE THAT  
 THIS WORD REPLACES ALL OCCURENCES OF THE SPECIFIED WORDS, EVEN  
 WITHIN COMMENTS OR CHARACTER STRINGS!
- FL2**  
 AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT NATURAL  
 LOGARITHM OF 2.0, THAT IS, 0.693147181.
- FL2E**  
 AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT LOGARITHM, BASE  
 2 OF E (THE BASE OF THE NATURAL LOGARITHMS, 2.71828...), THAT  
 IS, 1.44269504.

## FLD

A VARIABLE THAT ONE SETS TO THE TOTAL FIELD LENGTH DESIRED FOR NUMERIC OUTPUT. SEE DPL AND W.D.

## FLIT

STORE IN THE NEXT AVAILABLE DICTIONARY LOCATION THE ADDRESS OF THE ROUTINE WHICH PLACES FLOATING-POINT LITERALS ON THE STACK AT EXECUTION TIME.

## FLN

<FP-VALUE> FLN <FP-RESULT>  
COMPUTE THE NATURAL LOGARITHM OF <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.

## FLOATING (OLD)

SETS AN INTERNAL FLAG TO INDICATE THAT NUMBERS CONTAINING A PERIOD ARE TO BE INTERPRETED AS FLOATING-POINT NUMBERS, NOT AS DOUBLE-WORD INTEGERS. SEE DPREC.

## FLOG

<FP-VALUE> FLOG <FP-RESULT>  
COMPUTE THE LOGARITHM, BASE 10 OF <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.

## FLUSH

WRITE ALL BLOCKS THAT HAVE BEEN FLAGGED AS UPDATED TO DISC OR TAPE. SEE UPDATE.

## FMAX

<FP-VALUE1> <FP-VALUE2> FMAX <FP-RESULT>  
LEAVE THE GREATER OF <FP-VALUE1> AND <FP-VALUE2> ON THE STACK.

## FMIN

<FP-VALUE1> <FP-VALUE2> FMIN <FP-RESULT>  
LEAVE THE LESSER OF <FP-VALUE1> AND <FP-VALUE2> ON THE STACK.

## FMINUS

<FP-VALUE> FMINUS <FP-RESULT>  
NEGATE <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.

## FORGET

FORGET <NAME>  
DELETE THE DICTIONARY ENTRY FOR <NAME> AND ALL DICTIONARY ENTRIES FOLLOWING IT (I.E., EVERYTHING THAT HAS BEEN ENTERED INTO THE DICTIONARY AFTER THE DEFINITION OF <NAME>). THOUGH <NAME> MUST BE IN THE CONTEXT VOCABULARY, THE WORDS THAT FOLLOW IT IN THE DICTIONARY ARE DELETED, REGARDLESS WHICH VOCABULARY THEY BELONG TO. NORMALLY, FORGET SHOULD NOT BE USED WITHIN A COLON DEFINITION AS IT IS NOT A COMPILER DIRECTIVE. FOR SUCH APPLICATIONS THE WORD REMEMBER SHOULD BE USED.

## FORMATTER

LOADS THE DISC FORMATTING WORDS, WHICH ARE SELF EXPLANATORY. SEE UTIL.

- FORTH** (P)  
THE NAME OF THE PRIMARY VOCABULARY. EXECUTION OF THE WORD FORTH CAUSES FORTH TO BECOME THE CONTEXT VOCABULARY. SINCE FORTH CANNOT BE CHAINED TO ANY OTHER VOCABULARY, IT BECOMES THE ONLY VOCABULARY THAT WILL BE SEARCHED FOR DICTIONARY ENTRIES. UNLESS ADDITIONAL USER VOCABULARIES ARE DEFINED, NEW USER WORDS NORMALLY BECOME PART OF THE FORTH VOCABULARY.
- FPREAD** FPREAD <VALUE>  
READ THE FRONT PANEL SWITCH SETTING ON THE CPU AND LEAVE THE RESULTING 16-BIT VALUE ON THE STACK.
- FRATN** <FP-VALUE1> <FP-VALUE2> FRATN <FP-RESULT>  
COMPUTE THE ARCTANGENT (IN REVOLUTIONS) OF <FP-VALUE1> / <FP-VALUE2>, LEAVING THE RESULT ON THE STACK. NOTE THAT THIS WORD DOES NOT COMPUTE THE ARCTANGENT IN RADIANES BUT IN REVOLUTIONS.
- FREE**  
SETS THE VARIABLE FLD TO 0 AND THE VARIABLE DPL TO -1 WHICH SPECIFY THAT NUMERIC OUTPUT IS TO BE FREE FORMAT, THAT IS MINIMUM FIELD WIDTH AND NO RADIX POINT.
- FRESTORE**  
THIS WORD MUST BE EXECUTED BY AN INTERRUPT PROCESSING WORD WHICH HAS PREVIOUSLY EXECUTED FSAVE. FRESTORE WILL RESTORE ALL THE VARIABLES SAVED BY FSAVE.
- FSAVE**  
THIS WORD MUST BE EXECUTED BY AN INTERRUPT PROCESSING WORD PRIOR TO THE USE OF ANY FLOATING-POINT OPERATIONS. THIS WORD WILL SAVE ALL OF THE NON-REENTRANT VARIABLES USED BY THE FLOATING POINT WORDS. SEE FRESTORE.
- FSQRT** <FP-VALUE> FSQRT <FP-RESULT>  
COMPUTE THE SQUARE ROOT OF <FP-VALUE> AND LEAVE THE RESULT ON THE STACK.
- FW**  
INITIATE THE FORWARD SPACING OF A SINGLE MAG TAPE RECORD AND RETURN IMMEDIATELY. SEE FWSP.
- FWSP**  
INITIATE THE FORWARD SPACING OF A SINGLE MAG TAPE RECORD AND WAIT FOR THE OPERATION TO COMPLETE. SEE FW.



- G.                   <FP-VALUE> G.  
GENERALIZED FLOATING-POINT OUTPUT. OUTPUT THE <FP-VALUE> TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). IF THE VALUE IS EITHER TOO LARGE OR TOO SMALL TO BE OUTPUT BY F. THEN E. WILL BE USED.
- G?                   <ADDRESS> G?  
EQUIVALENT TO THE SEQUENCE "F@ G.".
- GCH                   GCH <CHAR-CODE>  
WAIT FOR A CHARACTER TO BE ENTERED ON THE TERMINAL AND PUSH ITS 7-BIT ASCII CODE ONTO THE STACK. SEE WCH. SEE APPENDIX A FOR A LISTING OF THE ASCII CODES.
- GO                   (OLD)  
LOAD BASIC FORTH (BLOCK 8) INTO THE DICTIONARY.
- GO-TO                <LINE#> GO-TO  
INTERRUPT INTERPRETATION OF THE CURRENT BLOCK AND RESUME INTERPRETATION STARTING WITH THE FIRST CHARACTER OF THE SPECIFIED LINE IN THE CURRENT BLOCK. THIS WORD MAY ONLY BE USED DURING THE LOADING OF A BLOCK. SEE IN.
- GODO                 (C)       GODO   WORD0 WORD1   ...   WORDN   THEN  
COMPUTED GO TO. WHEN THE WORD GODO IS EXECUTED THE VALUE ON TOP OF THE STACK SPECIFIES WHICH WORD IN THE SEQUENCE IS TO BE EXECUTED. IF THE VALUE <= 0 THEN WORD0 IS EXECUTED; IF VALUE = 1 THEN WORD1 IS EXECUTED; IF VALUE = 2 THEN WORD2 IS EXECUTED; ... IF VALUE >= N THEN WORDN IS EXECUTED.
- GS                     
PUTS THE 4010 IN GRAPHICS MODE AND PRODUCES A DARK VECTOR ON THE NEXT COMMAND. SEE US.

- H.                   <VALUE> H.  
HEXADECIMAL OUTPUT. OUTPUT <VALUE> AS A HEXADECIMAL NUMBER, UNSIGNED AND PRECEDED BY A BLANK ON THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE FORMAT SPECIFICATIONS GIVEN BY THE VARIABLES FLD AND DPL ARE OBSERVED. BASE IS NOT CHANGED.
- HBLOCK               <BLOCK#> HBLOCK  
READ A FORTH BLOCK FROM DISC. THIS WORD SHOULD BE USED IN PLACE OF THE WORD BLOCK IF THERE ARE HARDWARE ERRORS ON THE DISC. THE BLOCK IS READ BY SECTORS (TWO SECTORS COMPRISE EACH BLOCK) WITH UP TO FIVE RETRIES PER SECTOR, IN AN ATTEMPT TO RECOVER AS MUCH OF THE BLOCK AS POSSIBLE. THE DISC STATUS WORD IS OUTPUT IF ERRORS ARE ENCOUNTERED. NOTE THAT UNLIKE THE WORD BLOCK, HBLOCK DOES NOT RETURN AN ADDRESS ON THE STACK. INSTEAD, THE WORD PREV MAY BE USED TO OBTAIN THE MEMORY ADDRESS OF THE BLOCK.
- HEAD                 HEAD <ADDRESS>  
RETURNS A POINTER TO THE FIRST LOCATION OF THE LAST WORD DEFINED IN THE CURRENT VOCABULARY. HEAD IS EQUIVALENT TO THE SEQUENCE "CURRENT @".
- HERE                 HERE <ADDRESS>  
PUSH ONTO THE STACK THE ADDRESS OF THE NEXT AVAILABLE DICTIONARY LOCATION. SINCE THE DICTIONARY POINTER MAY BE AN INTERNAL REGISTER RATHER THAN A VARIABLE, IT SHOULD ONLY BE ACCESSED THROUGH THE WORDS HERE AND DP+!.
- HEX  
SET THE NUMERIC CONVERSION BASE TO HEXADECIMAL, THAT IS, SET THE VARIABLE BASE TO 16. SEE DECIMAL AND OCTAL.

**I** (C) **I** <VALUE>  
 PUSHES ONTO THE STACK THE CURRENT VALUE OF THE LOOP INDEX OF THE INNERMOST DO-LOOP CURRENTLY BEING EXECUTED. I MAY ONLY BE EXECUTED WITHIN THE WORD WHICH ACTUALLY EXECUTED THE DO-LOOP AND NOT FROM WITHIN SOME OTHER WORD WHICH HAS ITSELF BEEN EXECUTED FROM WITHIN THE DO-LOOP. SEE I'. THE FOLLOWING EXAMPLE WILL NOT WORK CORRECTLY:

```

: A ... I ... ;
: B ... DO ... A ... LOOP ... ;

```

THIS WORD MAY ALSO BE USED TO PUSH ONTO THE STACK A VALUE WHICH WAS PUSHED ONTO THE RETURN STACK USING >R. SEE >R AND R>.

**I'** (C) **I'** <VALUE>  
 PUSHES ONTO THE STACK THE CURRENT VALUE OF THE LOOP INDEX OF THE INNERMOST DO-LOOP CURRENTLY BEING EXECUTED IN THE WORD WHICH HAS CALLED THE WORD IN WHICH I' RESIDES. SINCE THE WORD I MAY NOT BE EXECUTED AT ANY LEVEL OTHER THAN WITHIN THE WORD WHICH EXECUTES THE DO-LOOP, I' GIVES ONE ACCESS TO THE LOOP PARAMETER AT ONE ADDITIONAL LEVEL. THE FOLLOWING EXAMPLE IS VALID:

```

: A ... I' ... ;
: B ... DO ... A ... LOOP ;

```

**I)** (A)  
 SETS THE VARIABLE MODE TO 7, SPECIFYING INDIRECT ADDRESSING FOR THE NEXT MEMORY REFERENCE INSTRUCTION. NOTE THAT THIS WORD DESIGNATES THE INSTRUCTION AS INDIRECT WHILE THE WORD 0) DESIGNATES AN ADDRESS AS INDIRECT.

**I.** <VALUE> <FIELD-WIDTH> **I.**  
 SINGLE-WORD INTEGER OUTPUT. CONVERT <VALUE> ACCORDING TO THE CURRENT NUMBER BASE AND OUTPUT IT TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL) USING THE SPECIFIED FIELD WIDTH. NO RADIX POINT IS PRINTED.

**I/O** (A) <VALUE> **I/O** <NAME>  
 DEFINE <NAME> AS A SINGLE-WORD MACHINE INSTRUCTION WHOSE BASIC MACHINE CODE REPRESENTATION IS <VALUE>. WHEN <NAME> IS EXECUTED THE TOP NUMBER ON THE STACK (USUALLY A DEVICE CODE, SHIFT COUNT OR REGISTER SPECIFICATION) IS INCLUSIVELY OR-ED WITH <VALUE> AND THE RESULT IS STORED IN THE NEXT AVAILABLE DICTIONARY LOCATION. SEE CPU, DL AND M/CPU.

- IAR, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN IAR INSTRUCTION  
(INCREMENT THE A REGISTER).
- IBR, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN IBR INSTRUCTION  
(INCREMENT THE B REGISTER).
- IC  
THE INTERPRETER INSTRUCTION COUNTER. A VARIABLE CONTAINING THE  
ADDRESS OF THE NEXT WORD TO EXECUTE.
- ID. <ADDRESS> ID.  
PRINT THE 3-CHARACTER/LENGTH IDENTIFIER OF THE WORD WHOSE  
DICTIONARY ENTRY STARTS AT THE SPECIFIED ADDRESS. USEFUL IN  
ANALYZING DUMPS.
- IF (C2+,P) <LOGICAL-VALUE> IF (TRUE-PART) ELSE (FALSE-PART) THEN  
<LOGICAL-VALUE> IF (TRUE-PART) THEN  
IF IS THE FIRST WORD OF A CONDITIONAL BRANCH. IF THE  
<LOGICAL-VALUE> IS TRUE (NON-ZERO) THE TRUE-PART WILL BE  
EXECUTED. IF THE <LOGICAL-VALUE> IS FALSE (ZERO) THE  
FALSE-PART, IF PRESENT, WILL BE EXECUTED. IF THE  
<LOGICAL-VALUE> IS FALSE AND THERE IS NO FALSE-PART SPECIFIED  
THE ENTIRE IF-THEN IS SKIPPED OVER AND EXECUTION RESUMES WITH  
THE WORDS FOLLOWING THE THEN. IF-ELSE-THEN CONDITIONALS MAY BE  
NESTED.
- IF, (A) <JUMP-CONDITION> IF, (TRUE-PART) ELSE, (FALSE-PART) THEN,  
<JUMP-CONDITION> IF, (TRUE-PART) THEN,  
IF, IS THE FIRST WORD OF A CONDITIONAL BRANCH. IF THE  
<JUMP-CONDITION> IS TRUE THEN THE SEQUENCE OF MACHINE  
INSTRUCTIONS COMPRISING THE TRUE-PART WILL BE EXECUTED. IF THE  
<JUMP-CONDITION> IS FALSE THE SEQUENCE OF MACHINE INSTRUCTIONS  
COMPRISING THE FALSE-PART WILL BE EXECUTED. IF THE  
<JUMP-CONDITION> IS FALSE AND THERE IS NO FALSE-PART SPECIFIED,  
THE ENTIRE IF,-THEN, IS SKIPPED OVER AND EXECUTION RESUMES WITH  
THE FIRST INSTRUCTION FOLLOWING THE THEN,. THE <JUMP-CONDITION>  
IS USUALLY SPECIFIED BY ONE OF THE WORDS A+, A-, A0, B0 OR OV.  
IF,-ELSE,-THEN, CONDITIONALS MAY BE NESTED.

- IFEND (E)  
TERMINATE A CONDITIONAL INTERPRETATION SEQUENCE BEGUN BY AN IFTRUE.
- IFTRUE (E) <LOGICAL-VALUE> IFTRUE ... OTHERWISE ... IFEND  
<LOGICAL-VALUE> IFTRUE ... IFEND  
THESE WORDS ARE SIMILAR TO THE IF-ELSE-THEN CONDITIONAL, HOWEVER THE IFTRUE-OTHERWISE-IFEND CONDITIONALS ARE TO BE USED DURING COMPILATION. ADDITIONALLY, UNLIKE THE IF-ELSE-THEN CONDITIONAL, THE IFTRUE-OTHERWISE-IFEND CONDITIONALS MAY NOT BE NESTED.
- IJMP, (A) <ADDRESS> <VALUE> IJMP,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN IJMP INSTRUCTION (INDEXED JUMP).
- IME, (A) <ADDRESS> <DEVICE-CODE> IME,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN IME INSTRUCTION (INPUT TO MEMORY).
- IMMEDIATE (OLD)  
MARK THE MOST RECENTLY CREATED DICTIONARY ENTRY SUCH THAT WHEN IT IS ENCOUNTERED AT COMPILE TIME IT WILL BE EXECUTED RATHER THAN COMPILED.
- IMOVE <SOURCE-ADDR> <DESTINATION-ADDR> IMOVE  
MOVE A GROUP OF SEQUENTIAL MEMORY CELLS, IN INVERSE ORDER, FROM THE <SOURCE-ADDR> TO THE <DESTINATION-ADDR>. THE LENGTH IS SPECIFIED BY <#CELLS>. INVERSE ORDER MEANS THAT THE LAST CELL IN THE SOURCE FIELD IS MOVED TO THE FIRST CELL OF THE DESTINATION FIELD, THE NEXT TO LAST CELL IN THE SOURCE FIELD IS MOVED TO THE SECOND CELL OF THE DESTINATION FIELD, ETC. SEE MOVE.
- IMP IMP <NAME>  
IF <NAME> IS THE NAME OF AN OVERLAY WHICH HAS PREVIOUSLY BEEN DEFINED AND SAVED THEN THIS WILL SET THE PRECEDENCE BIT OF THE OVERLAY, IDENTIFYING IT AS A VARIABLE OVERLAY. WHEN A VARIABLE OVERLAY IS IN THE MEMORY OVERLAY REGION AND ANOTHER OVERLAY IS TO BE READ IN, THE VARIABLE OVERLAY IS FIRST RE-WRITTEN TO DISC.
- IMP (OLD) IMP <NAME>  
TOGGLES THE PRECEDENT BIT OF THE SPECIFIED DICTIONARY ENTRY.
- IN  
A VARIABLE CONTAINING THE INDEX OF THE CHARACTER BEING INTERPRETED. ALTHOUGH THIS INDEX IS INITIALIZED AND INCREMENTED AUTOMATICALLY DURING INTERPRETATION, IT MAY BE MODIFIED TO AFFECT THE SEQUENCE OF INTERPRETATION. SEE GO-TO.

**INCLUDES**           <NAME1> INCLUDES <NAME2>  
 <NAME1> MUST BE THE NAME OF AN OVERLAY WHICH HAS PREVIOUSLY BEEN DEFINED AND SAVED; <NAME2> MUST BE THE NAME OF A WORD WHICH WAS DEFINED IN THAT OVERLAY. <NAME2> THEN BECOMES A DICTIONARY ENTRY IN MEMORY SO THAT EXECUTION OF <NAME2> WILL FIRST READ THE NEEDED OVERLAY INTO MEMORY AND THEN EXECUTE THE COPY OF <NAME2> IN THAT OVERLAY. REFERENCES MAY THEN BE MADE TO <NAME2> AS IF IT WERE PART OF THE DICTIONARY IN MEMORY. COMPARE THIS IMPLICIT LOADING OF AN OVERLAY WITH THE EXPLICIT LOADING PROVIDED BY O-LOAD. IF ANOTHER OVERLAY IS PRESENTLY IN MEMORY AND IF ITS PRECEDENCE BIT IS SET (SEE IMP) THEN IT WILL BE WRITTEN TO DISC BEFORE <NAME1> IS READ INTO MEMORY.

**INCR,**           (A)       <VALUE> INCR,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN INCR INSTRUCTION (INCREMENT AND COMBINE REGISTERS).

**INR,**           (A)       <ADDRESS> INR,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN INR INSTRUCTION (INCREMENT A WORD OF MEMORY).

**INTEGER**       (OLD)  
 RENAMED VARIABLE.

**INTX**           <VALUE> INTX  
 DEFINE A PSEUDO-VECTOR INTEGER VARIABLE, INITIALIZED TO <VALUE>. THE VALUE OF THIS VARIABLE MAY BE ACCESSED BY RCLX OR STORED BY STRX, AND IS ACCESSIBLE ONLY WITHIN A COLON DEFINITION. THESE WORDS ARE DESIGNED FOR CORE AND SPEED EFFICIENCY AND EACH INTX DECLARATION REQUIRES 3 DICTIONARY CELLS. SEE P-VX.

**IP**            (C)  
 A VARIABLE CONTAINING THE BYTE ADDRESS OF THE NEXT CHARACTER TO BE RETURNED BY CHFETCH. SEE COUNT, TYPE AND WRITE.

**ISR2**  
 A 2CONSTANT WHOSE VALUE IS THE DOUBLE-WORD FRACTION 2 \*\* -0.5.

**IAR,**           (A)  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN IAR INSTRUCTION (INCREMENT THE X REGISTER). NOTE THAT THE X REGISTER IS THE STACK POINTER IN VARIAN FORTH, THEREFORE THIS INSTRUCTION DEALLOCATES ONE WORD OF THE STACK.

- J** (C) J <VALUE>  
 WITHIN A NESTED DO-LOOP, THIS WORD PUSHES ONTO THE STACK THE CURRENT VALUE OF THE LOOP INDEX OF THE NEXT OUTER LOOP. J MAY ONLY BE EXECUTED WITHIN THE WORD WHICH ACTUALLY EXECUTED THE DO-LOOP AND NOT FROM WITHIN SOME OTHER WORD WHICH HAS ITSELF BEEN EXECUTED FROM WITHIN THE DO-LOOP. SEE I.
- JIF,** (A) <ADDRESS> <JUMP-CONDITION> JIF,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN JIF INSTRUCTION (CONDITIONAL JUMP). THE <JUMP-CONDITION> IS USUALLY SPECIFIED BY ONE OF THE WORDS A+, A-, AO, BO OR OV.
- JIFM,** (A) <ADDRESS> <JUMP-CONDITION> JIFM,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN JIFM INSTRUCTION (CONDITIONAL JUMP AND MARK). THE <JUMP-CONDITION> IS USUALLY SPECIFIED BY ONE OF THE WORDS A+, A-, AO, BO OR OV.
- JMP,** (A) <ADDRESS> JMP,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN JMP INSTRUCTION (UNCONDITIONAL JUMP).
- JMPM,** (A) <ADDRESS> JMPM,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN JMPM INSTRUCTION (UNCONDITIONAL JUMP AND MARK).
- JSR,** (A) <ADDRESS> <VALUE> JSR,  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN JSR INSTRUCTION (JUMP AND SET THE RETURN ADDRESS IN ONE OF THE REGISTERS).
- K** (C) K <VALUE>  
 WITHIN A NESTED DO-LOOP, THIS WORD PUSHES ONTO THE STACK THE CURRENT VALUE OF THE LOOP INDEX OF THE SECOND OUTER LOOP. K MAY ONLY BE EXECUTED WITHIN THE WORD WHICH ACTUALLY EXECUTED THE DO-LOOP AND NOT FROM WITHIN SOME OTHER WORD WHICH HAS ITSELF BEEN EXECUTED FROM WITHIN THE DO-LOOP. SEE I.
- KCURSOR** KCURSOR <Y-POSN> <X-POSN>  
 TURNS ON THE 4010 CROSS HAIR CURSORS AND WAITS FOR THE OPERATOR TO ENTER ANY CHARACTER. THREE SINGLE-WORD INTEGERS ARE RETURNED, THE X AND Y POSITIONS OF THE CURSORS AND THE ASCII CHARACTER CODE FOR THE CHARACTER THAT THE OPERATOR ENTERED.

## L2B10

AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT LOGARITHM, BASE 10 OF 2.0, THAT IS, 0.301029996.

LAL, (A) <SHIFT-COUNT> LAL,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LASL INSTRUCTION (LONG ARITHMETIC SHIFT LEFT).

LAR, (A) <SHIFT-COUNT> LAR,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LASR INSTRUCTION (LONG ARITHMETIC SHIFT RIGHT).

## LAST

A VARIABLE CONTAINING THE COMPILATION ADDRESS OF THE MOST RECENTLY MADE DICTIONARY ENTRY, WHICH MAY NOT YET BE A COMPLETE OR VALID ENTRY. IN ORDER TO EXECUTE A WORD RECURSIVELY, THE SEQUENCE "[P] : MYSELF LAST @ , ;" DEFINES THE WORD MYSELF WHICH MAY THEN BE USED WITHIN A COLON DEFINITION TO RECURSIVELY EXECUTE THE WORD BEING DEFINED.

LDA, (A) <ADDRESS> LDA,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LDA INSTRUCTION (LOAD THE A REGISTER FROM MEMORY).

LDB, (A) <ADDRESS> LDB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LDB INSTRUCTION (LOAD THE B REGISTER FROM MEMORY).

LDX, (A) <ADDRESS> LDX,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LDX INSTRUCTION (LOAD THE X REGISTER FROM MEMORY). NOTE THAT THE X REGISTER IS THE STACK POINTER IN VARIAN FORTH.

LEAVE (C)  
FORCE TERMINATION OF A DO-LOOP AT THE NEXT OPPORTUNITY BY SETTING THE LOOP LIMIT EQUAL TO THE CURRENT VALUE OF THE LOOP INDEX. THE VALUE OF THE INDEX REMAINS UNCHANGED AND EXECUTION CONTINUES THROUGH THE LOOP WITH THE TERMINATION OCCURRING AT THE NEXT EXECUTION OF EITHER LOOP OR +LOOP.

LENGTH <#POINTS> LENGTH  
SPECIFY THE NUMBER OF POINTS FOR AN FFT. <#POINTS> IS THE NUMBER OF REAL, DOUBLE-WORD INTEGER DATA POINTS FOR DFOURTRAN AND MUST BE A POWER OF 2. FOR DINVTRAN, (<#POINTS> / 2) + 1 COMPLEX, DOUBLE-WORD INTEGER DATA POINTS WILL YIELD <#POINTS> REAL, DOUBLE-WORD INTEGER POINTS.



- LINE**                   <LINE#> LINE <ADDRESS>  
PUSHES ONTO THE STACK THE ADDRESS IN MEMORY OF THE FIRST CHARACTER OF THE SPECIFIED LINE IN THE BLOCK WHOSE BLOCK NUMBER IS CONTAINED IN THE VARIABLE BLK.
- LINEIN**                LINEIN <ADDRESS>  
REQUEST A LINE OF INPUT FROM THE TERMINAL. THE LINE IS TERMINATED BY A CARRIAGE RETURN AND THE MEMORY ADDRESS OF THE CHARACTER STRING (WHICH IS STORED IN THE AVAILABLE DICTIONARY SPACE) IS RETURNED ON THE STACK.
- LINELOAD**             <LINE#> <BLOCK#> LINELOAD  
BEGIN INTERPRETING AT THE SPECIFIED LINE OF THE SPECIFIED BLOCK. THE SEQUENCE "<BLOCK#> LOAD" IS EQUIVALENT TO THE SEQUENCE "1 <BLOCK#> LINELOAD".
- LINEWRITE**           <LINE#> LINEWRITE  
OUTPUT TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL) THE SPECIFIED LINE (64 CHARACTERS) OF THE BLOCK WHOSE BLOCK NUMBER IS CONTAINED IN THE VARIABLE BLK.
- LIST**                 <BLOCK#> LIST  
LIST THE ASCII SYMBOLLIC CONTENTS OF THE SPECIFIED BLOCK ON THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL).
- LIT**                 (OLD,C)  
RENAMED /LIT/.
- LIT,**                (OLD)  
RENAMED /LIT/.
- LITERAL**  
THE WORD IN BASIC FORTH WHICH EITHER PUSHES A NUMBER ONTO THE STACK OR COMPILES IT INTO THE DICTIONARY, DEPENDING ON THE CURRENT STATE (COMPILING OR EXECUTING).
- LLRL,**               (A)       <SHIFT-COUNT> LLRL,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LLRL INSTRUCTION (LONG LOGICAL ROTATE LEFT).
- LLSR,**               (A)       <SHIFT-COUNT> LLSR,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LLSR INSTRUCTION (LONG LOGICAL SHIFT RIGHT).

- LOAD**                   <BLOCK#> LOAD  
BEGIN INTERPRETATION OF THE SPECIFIED BLOCK, STARTING WITH THE FIRST LINE IN THE BLOCK. THE BLOCK MUST TERMINATE ITS OWN INTERPRETATION WITH EITHER ;S, --> OR CONTINUED.
- LOADER**                <BLOCK#> LOADER <NAME>  
DEFINE THE WORD <NAME> WHICH, WHEN EXECUTED, WILL CAUSE THE SPECIFIED BLOCK TO BE LOADED.
- LOG2(10)**  
AN FCONSTANT WHOSE VALUE IS THE FLOATING-POINT LOGARITHM, BASE 2 OF 10.0, THAT IS, 3.32192809.
- LOOP**                (C)  
INCREMENT THE DO-LOOP INDEX BY +1, TERMINATING THE LOOP IF THE NEW VALUE OF THE INDEX IS EQUAL TO OR GREATER THAN THE LIMIT.
- L PLOT**                <X-POSN> <Y-POSN> L PLOT  
DRAWS A VECTOR ON THE 4010 TO THE LOGICAL POSITION SPECIFIED BY <X-POSN> AND <Y-POSN>. THIS NEW POSITION IS THEN SAVED IN /4010. <X-POSN> AND <Y-POSN> ARE FLOATING-POINT NUMBERS IN THE RANGE 0.0 - 1023.3 AND 0.0 - 780.0.
- LRL,**                (A)       <SHIFT-COUNT> LRL,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LRLA INSTRUCTION (LOGICAL ROTATE LEFT THE A REGISTER).
- LRLB,**               (A)       <SHIFT-COUNT> LRLB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LRLB INSTRUCTION (LOGICAL ROTATE LEFT THE B REGISTER).
- LS**                    <VALUE> <SHIFT-COUNT> LS <RESULT>  
ROTATE <VALUE> LOGICALLY LEFT OR RIGHT. IF THE <SHIFT-COUNT> IS POSITIVE THE SHIFT IS A LOGICAL ROTATE LEFT WHILE IF <SHIFT-COUNT> IS NEGATIVE THE SHIFT IS A LOGICAL ROTATE RIGHT. SEE 2LS.
- LSR,**                (A)       <SHIFT-COUNT> LSR,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LSRA INSTRUCTION (LOGICAL SHIFT RIGHT THE A REGISTER).
- LSRB,**               (A)       <SHIFT-COUNT> LSRB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN LSRB INSTRUCTION (LOGICAL SHIFT RIGHT THE B REGISTER).
- LWA?**                LWA? <ADDRESS>  
READ IN THE LAST-WORD-ADDRESS COUNTER OF THE MAG TAPE CONTROLLER AND PUSH THIS ADDRESS ONTO THE STACK.

- M\***                   <VALUE1> <VALUE2> M\* <DW-RESULT>  
MIXED PRECISION MULTIPLY, FORMING A DOUBLE-WORD PRODUCT FROM TWO SINGLE-WORD MULTIPLICANDS. SEE 2M\*.
- M+**                   <DW-VALUE> <VALUE> M+ <DW-RESULT>  
MIXED PRECISION ADDITION, ADDING THE SINGLE-WORD <VALUE> TO <DW-VALUE> FORMING A DOUBLE-WORD RESULT.
- M/**                   <DW-VALUE> <VALUE> M/ <QUOTIENT>  
MIXED PRECISION DIVIDE, DIVIDING THE <DW-VALUE> BY THE SINGLE-WORD <VALUE> FORMING A SINGLE-WORD <RESULT>. NOTE THAT THE QUOTIENT IS TRUNCATED AND ANY REMAINDER IS LOST. SEE M/MOD.
- M/CPU**           (A)       <VALUE> M/CPU <NAME>  
DEFINE <NAME> AS A MEMORY REFERENCE INSTRUCTION WHOSE BASIC MACHINE CODE REPRESENTATION IS <VALUE>. WHEN <NAME> IS EXECUTED THE TOP NUMBER ON THE STACK IS EITHER A MEMORY ADDRESS OR AN IMMEDIATE OPERAND AND THE CURRENT VALUE OF MODE DETERMINES THE ADDRESSING MODE OF THE INSTRUCTION. THE VALUE OF <MODE>, THE TOP NUMBER ON THE STACK AND THE LIMITATIONS OF THE VARIAN HARDWARE WILL DETERMINE WHETHER THE SINGLE-WORD OR DOUBLE-WORD VERSION OF THE INSTRUCTION IS GENERATED. THE INSTRUCTION WILL BE STORED IN THE NEXT AVAILABLE WORD(S) OF THE DICTIONARY. SEE CPU, DL AND I/O.
- M/MOD**           <DW-VALUE> <VALUE> M/MOD <REMAINDER> <QUOTIENT>  
MIXED PRECISION DIVIDE, DIVIDING THE <DW-VALUE> BY THE SINGLE-WORD <VALUE> YIELDING A SINGLE-WORD <QUOTIENT> ON TOP OF THE STACK AND A SINGLE-WORD <REMAINDER> BELOW. THE REMAINDER WILL HAVE THE SIGN OF THE DIVIDEND.
- MAPO**           (T)       MAPO <ADDRESS>  
THIS WORD PUSHES ONTO THE STACK THE ADDRESS OF THE FIRST LOCATION IN THE TAPE MAP.
- MARK**           <SYMBOL> <SIZE> MARK  
DRAWS A SYMBOL ON THE 4010 AT THE CURRENT POSITION. <SYMBOL> IS A SINGLE-WORD INTEGER VALUE WHICH IS INTERPRETED AS FOLLOWS:
- 1 - PLUS SIGN
  - 2 - CROSS
  - 3 - BOX
  - 4 - DIAMOND
- <SIZE> IS A SINGLE-WORD INTEGER THAT SPECIFIES THE SYMBOL SIZE IN POINTS (A STANDARD ALPHA CHARACTER IS 14 POINTS HIGH).
- MAX**           <VALUE1> <VALUE2> MAX <RESULT>  
LEAVE THE GREATER OF <VALUE1> AND <VALUE2> ON THE STACK.

MERG,           (A)           <VALUE> MERG,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN MERG INSTRUCTION (COMBINE  
REGISTERS).

MESSAGE           <DW-VALUE> MESSAGE  
A SINGLE LINE (64 CHARACTERS) OF A BLOCK IS OUTPUT TO THE  
CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL).  
<DW-VALUE> SPECIFIES BOTH THE BLOCK# AND THE LINE#, WITH  
<DW-VALUE> = (BLOCK# \* 100) + LINE#. IF <DW-VALUE> IS POSITIVE  
THEN A CARRIAGE RETURN WILL PRECEDE THE MESSAGE. A NEGATIVE  
<DW-VALUE> MAY BE USED TO SPECIFY NO CARRIAGE RETURN.

MIN               <VALUE1> <VALUE2> MIN <RESULT>  
LEAVE THE LESSER OF <VALUE1> AND <VALUE2> ON THE STACK.

MINUS            <VALUE> MINUS <RESULT>  
NEGATE <VALUE> BY TAKING ITS TWOS-COMPLEMENT.

- MOD**                   <VALUE1> <VALUE2> MOD <REMAINDER>  
CALCULATE <VALUE1> / <VALUE2> AND LEAVE ONLY THE REMAINDER ON THE STACK. THE REMAINDER WILL HAVE THE SIGN OF THE DIVIDEND.
- MODE**                (A)  
A VARIABLE WHICH SPECIFIES THE TYPE OF ADDRESSING TO BE USED FOR THE NEXT MEMORY REFERENCE INSTRUCTION. THE VALUES OF MODE ARE:
- 0 - DIRECT ADDRESSING (DEFAULT).
  - 1 - IMMEDIATE ADDRESSING. SEE #.
  - 4 - RELATIVE ADDRESSING (P REGISTER). SEE P).
  - 5 - INDEXING OFF THE X REGISTER. SEE X) AND S).
  - 6 - INDEXING OFF THE B REGISTER. SEE B) AND E).
  - 7 - INDIRECT ADDRESSING. SEE I).
- THE VALUE OF MODE IS RESET TO ZERO AFTER EVERY MEMORY REFERENCE INSTRUCTION IS COMPILED INTO THE DICTIONARY.
- MOVE**                <SOURCE-ADDR> <DESTINATION-ADDR> <#CELLS> MOVE  
MOVE A GROUP OF SEQUENTIAL MEMORY CELLS FROM THE <SOURCE-ADDRESS> TO THE <DESTINATION-ADDRESS>. THE LENGTH IS SPECIFIED BY <#CELLS>. AN OVERLAPPING OF DATA CAN OCCUR AND THE MOVE IS PERFORMED BY MOVING THE CONTENTS OF <SOURCE-ADDRESS> FIRST (SIMILAR TO THE IBM MVC INSTRUCTION). THIS ALLOWS ONE TO ZERO AN ENTIRE REGION OF N CELLS BY SETTING THE FIRST CELL TO ZERO AND THEN MOVING N-1 CELLS FROM THE FIRST CELL TO THE SECOND CELL. SEE IMOVE.
- MPY,**                (A)       <ADDRESS> MPY,  
AN ASSEMBLER MACRO WHICH GENERATES A SEQUENCE OF MACHINE INSTRUCTIONS TO MULTIPLY THE CONTENTS OF THE B REGISTER WITH THE CONTENTS OF THE SPECIFIED MEMORY LOCATION. THE SEQUENCE OF INSTRUCTIONS GENERATED IS TZA AND MUL.
- MS**                   <VALUE> MS  
DELAY FOR APPROXIMATELY <VALUE> MILLISECONDS (ACCURATE TO WITHIN 2 PERCENT). THIS WAIT IS AN INSTRUCTION LOOP, NOT BASED ON ANY EXTERNAL CLOCK AND THEREFORE ASSUMES THAT NO INTERRUPTS OR DMA ARE OCCURING SIMULTANEOUSLY.
- MSEC**                (OLD)  
RENAMED MS.
- MSGO**                (OLD)  
A VARIABLE CONTAINING THE BYTE-ADDRESS OF THE BEGINNING OF THE INPUT BUFFER.

- MTPERR**                    MTPERR <LOGICAL-VALUE>  
TEST THE MAG TAPE FOR A PARITY ERROR (AFTER A READ OR A WRITE) AND PUSH A <LOGICAL-VALUE> ONTO THE STACK CORRESPONDING TO THE PARITY ERROR FLAG.
- MTR**                      <ADDRESS> <#WORDS> MTR <ADDRESS> <#WORDS>  
INITIATE THE READING OF A MAG TAPE RECORD INTO THE BUFFER SPECIFIED BY <ADDRESS>. A MAXIMUM OF <#WORDS> WILL BE READ IN. RETURN IS MADE AS SOON AS THE OPERATION IS INITIATED. THE VARIABLE >BCD DETERMINES THE READING MODE OF THE 7-TRACK TAPE (BINARY OR BCD). NOTE THAT THIS WORD DOES NOT POP ITS TWO PARAMETERS OFF THE STACK. SEE MTREAD.
- MTREAD**                  <ADDRESS> <#WORDS> MTREAD  
EXECUTE THE WORD MTR AND WAIT FOR THE OPERATION TO COMPLETE. ERROR CHECKING IS PERFORMED AND IF A PARITY ERROR IS DETECTED THE READ WILL BE RETRIED UP TO 5 TIMES, AT WHICH TIME THE MESSAGE "PARITY" WILL BE OUTPUT AND THE OPERATION ABORTED.
- MTREJ**                    MTREJ <LOGICAL-VALUE>  
PUSH A <LOGICAL-VALUE> ONTO THE STACK DEPENDING ON WHETHER OR NOT THE LAST COMMAND TO THE MAG TAPE WAS REJECTED.
- MTW**                      <ADDRESS> <#WORDS> MTW <ADDRESS> <#WORDS>  
INITIATE THE WRITING OF A MAG TAPE RECORD FROM THE MEMORY ADDRESS SPECIFIED. <#WORDS> SPECIFIES THE NUMBER OF WORDS TO WRITE. RETURN IS MADE AS SOON AS THE OPERATION IS INITIATED. THE VARIABLE >BCD DETERMINES THE WRITING MODE OF THE 7-TRACK TAPE (BINARY OR BCD). NOTE THAT THIS WORD DOES NOT POP ITS PARAMETERS OFF THE STACK. SEE MTWRITE.
- MTWAIT**  
WAIT UNTIL THE MAG TAPE UNIT IS READY AND THEN RETURN.
- MTWRITE**                <ADDRESS> <#WORDS> MTWRITE  
EXECUTE THE WORD MTW AND WAIT FOR THE OPERATION TO COMPLETE. ERROR CHECKING IS PERFORMED AND IF A PARITY ERROR IS DETECTED THE WRITE WILL BE RETRIED UP TO 5 TIMES, AT WHICH POINT A THREE INCH SECTION OF TAPE WILL BE ERASED, THE OPERATION STARTED AGAIN AND THE MESSAGE "WRITE ERROR" OUTPUT.
- MUL,**                    (A)            <ADDRESS> MUL,  
THE ASSEMBLER MACRO FOR THE VARIAN MUL INSTRUCTION (MULTIPLY THE B REGISTER AND MEMORY THEN ADD IN THE A REGISTER).

N

A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF A REGION USED BY FORTH FOR TEMPORARY STORAGE. FOR EXAMPLE, THE SEQUENCE "N 6 + @" WILL PUSH ONTO THE STACK THE STATUS BITS FROM THE LAST DISC OPERATION, IF AN ERROR OCCURED.

N.

<VALUE> <FIELD-WIDTH> <#PLACES> N.  
 CONVERT <VALUE> ACCORDING TO THE CURRENT NUMBER BASE AND OUTPUT IT TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). FLD IS SET TO THE SPECIFIED FIELD WIDTH AND DPL IS SET TO THE SPECIFIED NUMBER OF DIGITS TO APPEAR TO THE RIGHT OF THE RADIX POINT.

NDROP

<VALUE> NDROP  
 VALUE SPECIFIES HOW MANY WORDS ARE TO BE DROPPED FROM THE TOP OF THE STACK. THE SEQUENCE "1 NDROP" IS EQUIVALENT TO DROP, "2 NDROP" IS EQUIVALENT TO 2DROP AND "3 NDROP" IS EQUIVALENT TO 3DROP.

NEXT

(A)  
 A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE IN FORTH THAT DOES NOTHING TO THE STACK. THE NORMAL SEQUENCE WOULD BE "NEXT JMP,".

NEWTAPE

CREATES AN EMPTY TAPE WITHOUT LOADING THE BLOCK HANDLERS. THE TAPE IS REWOUND AND AN END-OF-FILE IS WRITTEN. SEE UTIL.

NOP,

(A)  
 THE ASSEMBLER MNEMONIC FOR THE VARIAN NOP INSTRUCTION (NO-OPERATION).

NOT

(A) <JUMP-CONDITION> NOT <RESULT>  
 NEGATE <JUMP-CONDITION> WHICH IS ASSUMED TO BE A MACHINE JUMP CONDITION. SEE A+, A-, AO, BO AND OV. FOR EXAMPLE, THE SEQUENCE "<ADDRESS> AO NOT JIF," WILL JUMP TO <ADDRESS> ONLY IF THE A REGISTER IS NOT ZERO.

NUMBER

NUMBER <DW-RESULT>  
 CONVERT THE CHARACTER STRING WHICH WAS LEFT IN THE DICTIONARY BUFFER BY WORD AS A NUMBER, RETURNING THE DOUBLE-WORD RESULT ON THE STACK. IF THE CHARACTER STRING CONTAINS CHARACTERS WHICH ARE NOT VALID IN A NUMBER, A "?Q" ERROR WILL OCCUR. AFTER CONVERSION THE VARIABLE #D CONTAINS THE NUMBER OF DIGITS TO THE RIGHT OF THE RADIX POINT OR COMMA. IF THE CURRENT NUMBER BASE IS LESS THAN OR EQUAL TO 10 (DECIMAL) THEN A NUMBER TERMINATED BY THE CHARACTER B IS CONVERTED AS AN OCTAL NUMBER.

## O-BLK

A VARIABLE CONTAINING THE BLOCK NUMBER ON DISC OF WHERE THE NEXT OVERLAY REGION IS TO BE STORED. THE USER SHOULD STORE THE STARTING BLOCK NUMBER OF THEIR OVERLAY REGION ON DISC IN O-BLK PRIOR TO EXECUTING A-SAVE.

## O-DEFINE

<#CELLS> O-DEFINE <NAME>

DEFINES AN OVERLAY AREA IN THE DICTIONARY WHOSE LENGTH IS <#CELLS> AND WHOSE IDENTIFIER IS <NAME>. THE DICTIONARY POINTER IS ADVANCED BY <#CELLS>. THE NUMBER OF BLOCKS ON DISC THAT WILL BE REQUIRED BY EACH OVERLAY USING <NAME> WILL BE <#CELLS> / 512 (ROUNDED UP TO THE NEAREST INTEGER). EXECUTING <NAME> WILL MOVE THE DICTIONARY POINTER BACK TO THE BEGINNING OF THE OVERLAY REGION SO THAT SUBSEQUENTLY DEFINED WORDS, UP TO THE NEXT A-SAVE OR O-SAVE, WILL COMPRISE AN OVERLAY. AFTER AN OVERLAY HAS BEEN DEFINED AND SAVED, THE DICTIONARY POINTER IS RESTORED TO ITS VALUE PRECEDING THE EXECUTION OF <NAME>.

## O-LOAD

<NAME> O-LOAD

<NAME> MUST BE A PREVIOUSLY SAVED OVERLAY (SEE A-SAVE AND O-SAVE) WHICH IS THEN EXPLICITLY LOADED INTO MEMORY. COMPARE THIS EXPLICIT LOADING OF AN OVERLAY WITH THE IMPLICIT LOADING PROVIDED BY INCLUDES. IF ANOTHER OVERLAY IS PRESENTLY IN MEMORY AND IF ITS PRECEDENCE BIT IS SET (SEE IMP) THEN IT WILL BE WRITTEN TO DISC BEFORE <NAME> IS READ INTO MEMORY.

## O.

<VALUE> O.

OCTAL OUTPUT. OUTPUT <VALUE> AS AN OCTAL NUMBER, UNSIGNED AND PRECEDED BY A BLANK ON THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). THE FORMAT SPECIFICATIONS GIVEN BY THE VARIABLES FLD AND DPL ARE OBSERVED. BASE IS NOT CHANGED.

## OAR,

(A) <DEVICE-CODE> OAR,

THE ASSEMBLER MNEMONIC FOR THE VARIAN OAR INSTRUCTION (OUTPUT THE A REGISTER).

## OBR,

(A) <DEVICE-CODE> OBR,

THE ASSEMBLER MNEMONIC FOR THE VARIAN OBR INSTRUCTION (OUTPUT THE B REGISTER).

## OCTAL

SETS THE NUMERIC CONVERSION BASE TO OCTAL, THAT IS, SET THE VARIABLE BASE TO 8. SEE DECIMAL AND HEX.

## OFF!

<PUSHBUTTON#> OFF!

TURN OFF THE STATUS BIT OF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON. <PUSHBUTTON#> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PBARRAY, ON!, PSTOGGLE AND PTOGGLE.

## OFFLINE

FORCES THE MAG TAPE DRIVE OFF-LINE.



- OME, (A) <ADDRESS> <DEVICE-CODE> OME,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN OME INSTRUCTION (OUTPUT FROM MEMORY).
- ON! <PUSHBUTTON#> ON!  
TURN ON THE STATUS BIT OF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PBARRAY, OFF!, PSTOGGLE AND PTOGGLE.
- OP (C)  
A VARIABLE CONTAINING THE BYTE-ADDRESS OF THE NEXT CHARACTER TO BE PLACED IN CORE BY THE SUBROUTINE DEPOSIT. SEE O)S.
- OR <VALUE1> <VALUE2> OR <RESULT>  
COMPUTE THE BITWISE INCLUSIVE-OR OF <VALUE1> AND <VALUE2>, LEAVING THE RESULT ON THE STACK.
- ORA, (A) <ADDRESS> ORA,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ORA INSTRUCTION (INCLUSIVELY-OR MEMORY WITH THE A REGISTER).
- ORCX  
INITIATES AN ANONYMOUS CODE DEFINITION, PLACING ITS ADDRESS IN THE PSEUDO-VECTOR ENTRY X, FOR SUBSEQUENT ADOPTION BY THE WORD ADOX. SIMILAR TO THE ANONYMOUS COLON DEFINITIONS INITIATED BY :ORX. IT IS VERY IMPORTANT TO REMEMBER THAT FORTH'S COMPILATION FLAG IS NOT SET WHILE ASSEMBLING MACHINE CODE INSTRUCTIONS, THAT IS, FORTH REMAINS IN EXECUTION MODE. SEE P-VX.
- OTHERWISE (E)  
THIS WORD PRECEDES THE FALSE-PART OF AN INTERPRETER LEVEL CONDITIONAL. SEE IFTRUE.
- OV (A)  
A CONSTANT WHOSE VALUE SPECIFIES THE JUMP CONDITION FOR THE "OVERFLOW SET" TEST. USUALLY FOLLOWED BY IF, END, JIF, JIFM, OR XIF,. REFER TO PAGE 20-18 OF THE VARIAN HANDBOOK. SEE NOT.
- OV! (A)  
MODIFY THE PREVIOUS WORD IN THE DICTIONARY (ASSUMED TO BE A REGISTER CHANGE INSTRUCTION) TO BE EXECUTED ONLY IF THE OVERFLOW BIT IS ON (I.E. - BIT 8 OF THE PREVIOUS WORD IN THE DICTIONARY IS TURNED ON). REFER TO PAGE 20-33 OF THE VARIAN HANDBOOK.
- OVER <VALUE1> <VALUE2> OVER <VALUE1> <VALUE2> <VALUE1>  
PUSH A COPY OF <VALUE1> ONTO THE TOP OF THE STACK, WITHOUT REMOVING ANY WORDS FROM THE STACK.

P)

(A)

SETS THE VARIABLE MODE TO 4, SPECIFYING RELATIVE ADDRESSING OFF THE P REGISTER FOR THE NEXT MEMORY REFERENCE INSTRUCTION.

P-VX

P-VX &lt;ADDRESS&gt;

PUSHES ONTO THE STACK THE ADDRESS OF ONE ELEMENT IN A 16-WORD TEMPORARY STORAGE AREA KNOWN AS THE PSEUDO-VECTOR TABLE. THE FOURTH CHARACTER OF THE WORD P-VX SPECIFIES WHICH OF THE SIXTEEN ELEMENTS IS BEING REFERENCED: THE LOWER 4-BITS OF THIS CHARACTER, A VALUE BETWEEN 0 AND 15, IS USED, IMPLYING THAT FOR EXAMPLE, P-V1 AND P-VA BOTH REFERENCE THE SECOND ELEMENT. THESE SIXTEEN TEMPORARY LOCATIONS ARE USED DURING COMPILATION EITHER DIRECTLY FOR TEMPORARY STORAGE BY THE PROGRAMMER OR INDIRECTLY BY THE WORDS :ORX, <<LX, >>LX, ADOX, INTX, ORCX, RCLX AND STRX. THE EQUIVALENCE BETWEEN THE LOWER 4-BITS OF THE CHARACTER X ARE AS FOLLOWS:

|    |   |   |   |   |         |
|----|---|---|---|---|---------|
| 00 | = | 0 | @ | P |         |
| 01 | = | 1 | A | Q | !       |
| 02 | = | 2 | B | R | "       |
| 03 | = | 3 | C | S | #       |
| 04 | = | 4 | D | T | \$      |
| 05 | = | 5 | E | U | PERCENT |
| 06 | = | 6 | F | V | &       |
| 07 | = | 7 | G | W | '       |
| 10 | = | 8 | H | X | (       |
| 11 | = | 9 | I | Y | )       |
| 12 | = | : | J | Z | *       |
| 13 | = | ; | K | [ | +       |
| 14 | = | < | L | \ | ,       |
| 15 | = | = | M | ] | -       |
| 16 | = | > | N | ^ | .       |
| 17 | = | ? | O | _ | /       |

PACK

&lt;DW-VALUE&gt; PACK &lt;DW-RESULT&gt;

CONVERT <DW-VALUE> FROM A VARIAN DOUBLE-WORD INTEGER TO A PACKED, CAMAC 24-BIT VALUE, IN PREPARATION FOR CAMAC OUTPUT. SEE UNPK.

PAGE

ERASES THE OPERATOR'S TERMINAL SCREEN OR SOME SIMILAR ACTION APPROPRIATE FOR THE PARTICULAR DEVICE.

**PB#** A CONSTANT WHOSE VALUE SPECIFIES THE PUSHBUTTON NUMBER OF THE CAMAC DISPLAY PANEL PUSHBUTTON THAT WAS PRESSED TO GENERATE AN INTERRUPT. THE VALUE OF PB# WILL BE IN THE RANGE 1 THROUGH 31 AND SHOULD BE EXAMINED ONLY WHEN THE VARIABLE PB@ IS NON-ZERO (TRUE). SEE PBENABLE AND PBDISABLE.

**PB@** PB@ <LOGICAL-RESULT>  
PUSH A <LOGICAL-RESULT> ONTO THE STACK DEPENDING ON WHETHER A CAMAC DISPLAY PANEL PUSHBUTTON INTERRUPT HAS OCCURED. IF AN INTERRUPT HAS OCCURED, THE CONSTANT PB# WILL CONTAIN THE NUMBER OF THE PUSHBUTTON THAT WAS PRESSED. SEE PBENABLE AND PBDISABLE.

**PBARRAY** A 32-WORD VECTOR WHICH MUST BE SET BY THE USER TO CONTAIN THE STATUS BIT, INITIAL LIGHT STATUS AND TOGGING BITS FOR THE CAMAC DISPLAY PANEL PUSHBUTTON LIGHTS. THE MOST SIGNIFICANT BIT OF EACH WORD IN PBARRAY IS THE PUSHBUTTON'S STATUS BIT, THE NEXT 7 BITS ARE UNUSED, THE NEXT 4 BITS SPECIFY THE INITIAL LIGHT STATUS FOR EACH OF THE PUSHBUTTON'S FOUR LIGHTS AND THE LOWER 4 BITS SPECIFY THE TOGGING BITS FOR EACH OF THE PUSHBUTTON'S FOUR LIGHTS. WHEN THE LIGHTS ARE TOGGLED THE NEW LIGHT STATUS EQUALS THE CURRENT LIGHT STATUS EXCLUSIVELY OR-ED WITH THE TOGGLE BITS. SEE PLARRAY, PBLIGHTS, PLTOGGLE, PSTOGGLE, PTOGGLE, ON! AND OFF!.

**PBDISABLE** DISABLE INTERRUPTS FROM THE CAMAC DISPLAY PANEL PUSHBUTTONS. SEE PBENABLE.

**PBENABLE** ENABLE INTERRUPTS FROM THE CAMAC DISPLAY PANEL PUSHBUTTONS. WHEN AN INTERRUPT OCCURS THE VARIABLE PB@ WILL BE SET NON-ZERO (TRUE) AND THE CONSTANT PB# WILL BE SET TO THE PUSHBUTTON NUMBER THAT WAS PRESSED (AN INTEGER VALUE IN THE RANGE 1 THROUGH 31). IT IS UP TO THE PROGRAM TO CONTINUALLY TEST PB@. NOTE THAT PUSHBUTTON NUMBER 0 IS HANDLED SPECIALLY BY FORTH AS A "?Z" ABORT. SEE PBDISABLE.

**PBLIGHTS** <LIGHT-BITS> <PUSHBUTTON#> PBLIGHTS  
TURN ON THE SPECIFIED LIGHTS IN THE DESIGNATED CAMAC DISPLAY PANEL PUSHBUTTON. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. <LIGHT-BITS> IS AN INTEGER VALUE WHOSE LOWER 4 BITS SPECIFY WHICH LIGHTS IN THE PUSHBUTTON ARE TO BE TURNED ON. SEE PLARRAY.

**PICK** <INDEX> PICK <RESULT>  
<INDEX> SPECIFIES A LOCATION ON THE STACK (1 SPECIFIES THE TOP OF THE STACK, 2 IS THE NEXT WORD ON THE STACK, ETC) AND A COPY OF THIS WORD IS PUSHED ONTO THE TOP OF THE STACK. THE SEQUENCE "2 PICK" IS EQUIVALENT TO THE WORD OVER.

## PLARRAY

A VECTOR MAINTAINED BY THE CAMAC DISPLAY PANEL ROUTINES TO REFLECT THE CURRENT STATUS OF EACH LIGHT OF EVERY PUSHBUTTON. A 4-BIT ENTRY IS REQUIRED FOR EACH PUSHBUTTON, WITH THE 4-BITS SPECIFYING (IN DECREASING ORDER OF SIGNIFICANCE): UPPER LEFT LIGHT, UPPER RIGHT LIGHT, LOWER LEFT LIGHT AND LOWER RIGHT LIGHT. FOUR OF THESE 4-BIT ENTRIES ARE PACKED TO EACH WORD OF PLARRAY WITH WORD 0 CONTAINING THE LIGHT STATUS FOR PUSHBUTTONS 0, 1, 2 AND 3, WORD 1 CONTAINING THE LIGHT STATUS FOR PUSHBUTTONS 4, 5, 6 AND 7, ETC. SEE PBARRAY, PBLIGHTS, PLTOGGLE AND PTOGGLE.

PLT                   <X-POSN> <Y-POSN> PLT  
 DRAWS A VECTOR ON THE 4010 TO THE PHYSICAL POSITION SPECIFIED BY <X-POSN> AND <Y-POSN> AND SAVES THIS NEW POSITION IN /4010. <X-POSN> AND <Y-POSN> ARE SINGLE-WORD INTEGERS IN THE RANGE 0-1023 AND 0-780.

PLTOGGLE            <PUSHBUTTON#> PLTOGGLE  
 TOGGLE THE LIGHT BITS OF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON. THE NEW VALUE OF THE LIGHT BITS ARE STORED IN PLARRAY. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PLARRAY AND PTOGGLE.

POP                (A)  
 A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE IN FORTH TO POP ONE WORD OFF THE STACK. THE SEQUENCE "POP 1-" LEAVES THE ADDRESS OF THE INTERPRETER ROUTINE TO POP TWO WORDS OFF THE STACK. THE NORMAL SEQUENCE IS EITHER "POP JMP," OR "POP 1- JMP,".

P PLOT             <X-POSN> <Y-POSN> P PLOT  
 DRAWS A VECTOR ON THE 4010 TO THE PHYSICAL POSITION SPECIFIED BY <X-POSN> AND <Y-POSN>. THIS NEW POSITION IS THEN SAVED IN /4010. <X-POSN> AND <Y-POSN> ARE FLOATING-POINT NUMBERS IN THE RANGE 0.0 - 1023.3 AND 0.0 - 780.0.

PREV  
 A VARIABLE CONTAINING THE ADDRESS OF THE LAST-WORD OF THE CURRENT BLOCK BUFFER BEING USED. THE SEQUENCE "PREV @ @" PUSHES ONTO THE STACK THE BLOCK NUMBER AND UPDATE BIT (MOST SIGNIFICANT BIT) OF THIS BLOCK. THE SEQUENCE "PREV 1+ @ @" PUSHES ONTO THE STACK THE BLOCK NUMBER AND UPDATE BIT OF THE ALTERNATE BLOCK.

## PRINTER

SETS A FLAG DENOTING THAT OUTPUT IS TO BE DIRECTED TO THE LINE PRINTER RATHER THAN THE TERMINAL. SEE TERMINAL.

## PRINTERS

DEFINES THE FOLLOWING WORDS WHICH, WHEN EXECUTED, LOAD THE WORDS FOR THE CORRESPONDING LINE PRINTER (SEE UTIL):

|       |                              |
|-------|------------------------------|
| CEN   | CENTRONICS                   |
| GOULD | GOULD                        |
| TI    | TEXAS INSTRUMENTS SILENT 700 |
| T40   | TELETYPE MODEL 40            |
| PTC   | PRINTEC                      |

## PSTOGGLE

<PUSHBUTTON#> PSTOGGLE

TOGGLE THE STATUS BIT OF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PBARRAY, ON!, OFF! AND PTOGGLE.

## PTOGGLE

<PUSHBUTTON#> PTOGGLE

TOGGLE THE STATUS BIT AND THE LIGHT STATUS OF THE SPECIFIED CAMAC DISPLAY PANEL PUSHBUTTON. <PUSHBUTTON> IS AN INTEGER VALUE IN THE RANGE 0 THROUGH 31. SEE PLARRAY, PBARRAY, PLTOGGLE, PSTOGGLE, ON! AND OFF!.

## PUSH

(A)

A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE IN FORTH TO PUSH THE CONTENTS OF THE A REGISTER ONTO THE STACK. THE NORMAL SEQUENCE IS "PUSH JMP,".

## PUT

(A)

A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF THE INTERPRETER ROUTINE IN FORTH TO POP ONE WORD FROM THE STACK AND THEN PUSH THE CONTENTS OF THE A REGISTER ONTO THE STACK. THE NORMAL SEQUENCE IS "PUT JMP,".

## QBF

(A)

A VARIABLE WHOSE VALUE IS THE BYTE ADDRESS OF THE BUFFER TO BE USED FOR OPERATOR INPUT.

## QUIT

CLEAR THE RETURN STACK (IN CASE THIS WORD IS EXECUTED FROM A WORD WHICH HAS BEEN CALLED BY OTHER WORDS) AND RETURN CONTROL TO THE TERMINAL.

## QUIT

(OLD)

RENAMED ABORT.

- R#**                    **R# <VALUE>**  
RANDOM NUMBER GENERATOR, RETURNING <VALUE> AS THE NEXT RANDOM NUMBER IN THE PSEUDO-RANDOM SEQUENCE. THE ALGORITHM USED IS A LINEAR CONGRUENTIAL SEQUENCE WITH A PERIOD OF 65536. <VALUE> WILL BE IN THE RANGE -32768 THROUGH 32767. IF A NUMBER SMALLER THAN <VALUE> IS REQUIRED THEN THE HIGH ORDER BITS OF <VALUE> SHOULD BE USED AS THE LOW ORDER BITS WILL BE MUCH LESS RANDOM THAN THE HIGH ORDER BITS.
- R#VALUE**  
A VARIABLE WHOSE VALUE IS THE PREVIOUS PSEUDO-RANDOM NUMBER GENERATED BY R#. IF A REPEATABLE SEQUENCE OF NUMBERS IS DESIRED, THEN ONE MAY SET R#VALUE TO ANY DESIRED VALUE, PRIOR TO EXECUTING R# FOR THE FIRST TIME.
- R1-2**                    **<VALUE> R1-2 <RESULT>**  
<VALUE> MUST BE A 14-BIT FRACTION AND THE SQUARE ROOT OF  $(1 - \langle \text{VALUE} \rangle ** 2)$  IS COMPUTED AND LEFT AS THE RESULT.
- R>**                    **(C)**  
POP THE TOP VALUE FROM THE RETURN STACK AND PUSH IT ONTO THE REGULAR STACK. SEE >R.

- RCLX** (C) RCLX <VALUE>  
PUSH ONTO THE STACK THE VALUE AT THE LOCATION ESTABLISHED BY INTX. SEE INTX, STRX AND P-VX.
- READ-MAP** (T)  
READ FROM THE CURRENT POSITION ON TAPE TO THE NEXT FILE-MARK, CONSTRUCTING IN MEMORY THE MAP RELATING THE PHYSICAL BLOCK POSITIONS ON TAPE WITH THE LOGICAL BLOCK NUMBER. THE TAPE SHOULD NORMALLY BE REWOUND PRIOR TO EXECUTING READ-MAP (SEE REWIND).
- REAL** <FP-VALUE> REAL <NAME>  
DEFINE A FLOATING-POINT VARIABLE. THE VALUE OF THE VARIABLE IS INITIALIZED TO <FP-VALUE> AND WHEN THE WORD <NAME> IS EXECUTED THE ADDRESS OF THE FLOATING-POINT VALUE OF THE VARIABLE WILL BE PUSHED ONTO THE STACK. SEE VARIABLE AND 2VARIABLE.
- RECOVER**  
READS A 32K CORE IMAGE FROM DISC TO MEMORY, EFFECTIVELY REPLACING THE ENTIRE MEMORY WITH WHATEVER THE MEMORY CONTAINED WHEN THE CORE IMAGE WAS WRITTEN. SEE SNAPSHOT.
- REDEF**  
SEARCHES EACH VOCABULARY IN THE DICTIONARY FOR REDEFINITIONS. SEE UTIL.
- REMEMBER** REMEMBER <NAME>  
DEFINE A WORD <NAME> WHICH WHEN EXECUTED WILL CAUSE ALL SUBSEQUENTLY DEFINED WORDS TO BE DELETED FROM THE DICTIONARY. THE WORD <NAME> MAY BE COMPILED INTO AND EXECUTED FROM A COLON DEFINITION. THE SEQUENCE "DISCARD REMEMBER DISCARD" PROVIDES A STANDARDIZED PREFACE TO ANY GROUP OF TRANSIENT WORDS. SEE FORGET.
- REPEAT** (C2-,P)  
EFFECT AN UNCONDITIONAL JUMP BACK TO THE BEGINNING OF A BEGIN-WHILE-REPEAT LOOP. SEE BEGIN.
- REPLACE** REPLACE <WORD1> <WORD2>  
REPLACE ALL OCCURENCES OF <WORD1> BY <WORD2> WHEN THE WORD FIX IS EXECUTED. NEITHER <WORD1> NOR <WORD2> MAY CONTAIN ANY SPACES. SEE FIX, \$REPLACE AND WINIT.
- RESIDENT**  
LOADS BLOCK 200. THE ACTIONS PERFORMED BY BLOCK 200 ARE TOTALLY USER DEPENDENT.

- RETURN** (A) <ADDRESS> RETURN  
AN ASSEMBLER MACRO TO GENERATE A JUMP INDIRECT INSTRUCTION TO THE SPECIFIED MEMORY <ADDRESS>. THIS WORD IS TYPICALLY USED TO RETURN FROM A MACHINE LANGUAGE SUBROUTINE, IN WHICH CASE <ADDRESS> IS PUSHED ONTO THE STACK BY EXECUTING THE NAME OF THE SUBROUTINE. SINCE THE RETURN ADDRESS OF A VARIAN SUBROUTINE IS STORED IN THE FIRST WORD OF THE SUBROUTINE, A JUMP INDIRECT TO THE FIRST WORD WILL RETURN CONTROL TO THE CALLER. SEE SUBROUTINE.
- REW**  
INITIATES A REWIND OF THE MAG TAPE AND WAITS FOR THE OPERATION TO COMPLETE. SEE RW.
- REWIND** (T)  
REWIND THE TAPE TO ITS LOAD POINT AND SET THE VARIABLE CUR TO 1.
- ROF,** (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN ROF INSTRUCTION (RESET THE OVERFLOW BIT).
- ROLL** <INDEX> ROLL  
<INDEX> SPECIFIES A LOCATION ON THE STACK (1 SPECIFIES THE TOP OF THE STACK, 2 IS THE NEXT WORD ON THE STACK, ETC) AND THIS WORD ON THE STACK IS MOVED TO THE TOP OF THE STACK WITH ALL WORDS ON THE STACK BETWEEN BEING MOVED DOWN ONE POSITION. THE SEQUENCE "1 ROLL" IS A NULL OPERATION, THE SEQUENCE "3 ROLL" IS EQUIVALENT TO ROT AND THE SEQUENCE "0 ROLL" IS UNDEFINED.
- ROT** <VALUE1> <VALUE2> <VALUE3> ROT  
<VALUE2> <VALUE3> <VALUE1>  
ROTATE THE TOP THREE POSITIONS ON THE STACK. <VALUE1> IS MOVED TO THE TOP OF THE STACK, <VALUE3> MOVES FROM THE TOP TO THE SECOND POSITION AND <VALUE2> MOVES FROM THE SECOND POSITION TO THE THIRD.
- RP** RP <ADDRESS>  
PUSHES ONTO THE STACK THE ADDRESS OF THE NEXT AVAILABLE LOCATION ON THE RETURN STACK.
- RS.C** <VALUE> RS.C <RESULT1> <RESULT2>  
<VALUE> MUST BE A 14-BIT FRACTION (REVOLUTIONS) AND IT IS REPLACED BY ITS SINE AND COSINE (IN RADIANS), BOTH 14-BIT FRACTIONS, WITH THE COSINE ON TOP OF THE STACK AND THE SINE BELOW.
- RW**  
INITIATES A REWIND OF THE MAG TAPE AND RETURNS IMMEDIATELY. SEE REW.



- S) (A)  
SETS THE VARIABLE MODE TO 5 AND PUSHES A VALUE OF ZERO ONTO THE STACK (FOR THE NEXT MEMORY REFERENCE INSTRUCTION). THE VALUE OF ZERO SPECIFIES THE DISPLACEMENT AND THE WORD S) THEREFORE SPECIFIES THAT THE NEXT MEMORY REFERENCE INSTRUCTION IS TO ADDRESS THE TOP WORD ON THE STACK (SINCE FORTH KEEPS THE ADDRESS OF THE TOP ELEMENT ON THE STACK IN THE X REGISTER).
- S. <VALUE> S.  
CONVERT <VALUE> ACCORDING TO THE CURRENT NUMBER BASE AND OUTPUT IT TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). FORMATTING OF THE NUMBER IS SPECIFIED BY THE VALUES OF THE VARIABLES FLD AND DPL.
- S@ (OLD)  
RENAMED PICK.
- SASK <VALUE>  
REQUEST THE INPUT OF A SINGLE-WORD VALUE FROM THE TERMINAL.
- SAVEDISK <BLOCK#> SAVEDISK  
TRANSFERS BLOCKS 1 THROUGH <BLOCK#> FROM DISC TO TAPE, EFFECTIVELY SAVING THE BLOCKS ON TAPE. <BLOCK#> IS OPTIONAL AND IF NOT SPECIFIED A VALUE OF 511 IS USED.
- SAVER (A)  
A CONSTANT WHOSE VALUE IS THE MEMORY ADDRESS OF A SUBROUTINE THAT WILL SAVE THE SYSTEM'S STATUS ON THE STACK AND THEN START INTERPRETATION OF A SEQUENCE OF HIGH LEVEL WORDS. THE WORDS SAVER AND UNSAVER, ARE USED TO EXECUTE SOME HIGH LEVEL FORTH WORDS FROM WITHIN A MACHINE CODE INTERRUPT PROCESSING WORD. THIS SUBROUTINE MUST BE CALLED WITH A JSR INSTRUCTION USING THE B REGISTER TO HOLD THE RETURN ADDRESS. THE STANDARD CALLING SEQUENCE WOULD BE "SAVER 6 JSR,". THIS SUBROUTINE WILL SAVE THE A REGISTER, OVERFLOW BIT, INTERPRETER INSTRUCTION COUNTER, FOREGROUND ADDRESS AND THE VARIABLE <T> (THE B REGISTER MUST BE SAVED BY THE CALLER BEFORE EXECUTING THE JSR). THE DICTIONARY LOCATIONS FOLLOWING THE JSR INSTRUCTION MUST CONTAIN THE COMPILATION ADDRESSES OF A SEQUENCE OF FORTH WORDS WHICH WILL THEN BE EXECUTED, AFTER THE SYSTEM STATUS HAS BEEN SAVED. THE FINAL WORD IN THE SEQUENCE MUST BE UNSAVER, WHICH WILL RESTORE THE SYSTEM STATUS THAT WAS SAVED BY SAVER AND THEN START EXECUTION OF THE MACHINE CODE FOLLOWING UNSAVER, IN THE DICTIONARY (USUALLY SOME CODE TO RESTORE THE B REGISTER AND RETURN FROM THE INTERRUPT).

- SEN, (A) <FUNCTION-DEVICE> SEN,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN SEN INSTRUCTION (SENSE A DEVICE).
- SENSE (A) <CONDITION-DEVICE> SENSE  
AN ASSEMBLER MACRO WHICH GENERATES A MACHINE INSTRUCTION. THE INSTRUCTION GENERATED IS AN SEN, (USING THE SPECIFIED <CONDITION-DEVICE>) WITH AN ADDRESS OF HERE+4. IT IS THEREFORE ASSUMED THAT A JUMP INSTRUCTION FOLLOWS THE SENSE INSTRUCTION AND IF THE SPECIFIED <CONDITION-DEVICE> IS TRUE THE JUMP INSTRUCTION IS BYPASSED, AND IF FALSE, THE JUMP INSTRUCTION IS EXECUTED.
- SET <VALUE> <ADDRESS> SET <NAME>  
DEFINE THE WORD <NAME> SUCH THAT WHEN IT IS EXECUTED, <VALUE> WILL BE STORED IN THE MEMORY LOCATION POINTED TO BY <ADDRESS>.
- SFIX <FP-VALUE> SFIX <RESULT>  
TRUNCATE <FP-VALUE> TO A SINGLE-WORD INTEGER VALUE. IF ONE WANTS TO ROUND THE FLOATING-POINT VALUE PRIOR TO TRUNCATION, THE FLOATING-POINT VALUE 0.5 SHOULD BE ADDED TO <FP-VALUE> PRIOR TO EXECUTING SFIX. SEE DFIX.
- SFLOAT <VALUE> SFLOAT <FP-RESULT>  
CONVERT THE SINGLE-WORD <VALUE> TO ITS FLOATING-POINT REPRESENTATION AND LEAVE THE RESULT ON THE STACK.
- SIN.COS <VALUE> SIN.COS <RESULT1> <RESULT2>  
<VALUE> IS A SINGLE-WORD INTEGER (MINUTES OF ARC) AND ITS SINE AND COSINE (BOTH 14-BIT FRACTIONS) ARE CALCULATED AND LEFT ON THE STACK WITH THE COSINE ON TOP OF THE STACK AND THE SINE BELOW.
- SKIPF  
INITIATE THE FORWARD SPACING OF THE MAG TAPE ONE FILE AND WAIT FOR THE OPERATION TO COMPLETE.
- SNAPSHOT  
WRITES AN ENTIRE CORE IMAGE (32K WORDS) ONTO THE DISC IN BLOCKS 2383 THROUGH 2447 (THE INNERMOST CYLINDERS OF THE REMOVABLE PLATTER). THIS CORE IMAGE MAY THEN BE RELOADED AT A LATER TIME USING THE WORD RECOVER.
- SOF, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN SOF INSTRUCTION (SET THE OVERFLOW BIT).

- SP                    SP <ADDRESS>  
PUSHES ONTO THE STACK THE ADDRESS OF THE TOPMOST STACK VALUE.
- SPACE  
OUTPUT A SINGLE SPACE TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL).
- SPACES                <VALUE> SPACES  
OUTPUT A STRING OF SPACES (BLANKS) TO THE CURRENT OUTPUT DEVICE. THE NUMBER OF SPACES IS SPECIFIED BY <VALUE>.
- SQRT                  <DW-VALUE> SQRT <RESULT>  
COMPUTE THE SQUARE ROOT OF <DW-VALUE> AND LEAVE THE SINGLE-WORD RESULT ON THE STACK.
- SRE,                  (A)            <ADDRESS> <VALUE> SRE,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN SRE INSTRUCTION (SKIP IF REGISTER EQUAL).
- STA,                  (A)            <ADDRESS> STA,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN STA INSTRUCTION (STORE THE A REGISTER IN MEMORY).
- STB,                  (A)            <ADDRESS> STB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN STB INSTRUCTION (STORE THE B REGISTER IN MEMORY).
- STRX                  <VALUE> STRX  
STORE <VALUE> AT THE LOCATION ESTABLISHED BY THE PREVIOUS INTX DEFINITION. SEE INTX, RCLX AND P-VX.
- STX,                  (A)            <ADDRESS> STX,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN STX INSTRUCTION (STORE THE X REGISTER IN MEMORY).
- SUB,                  (A)            <ADDRESS> SUB,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN SUB INSTRUCTION (SUBTRACT MEMORY FROM THE A REGISTER).
- SUBROUTINE            SUBROUTINE <NAME>  
CREATES A DICTIONARY ENTRY FOR THE NAMED MACHINE LANGUAGE SUBROUTINE. ASSEMBLER BECOMES THE CONTEXT VOCABULARY. <NAME> BECOMES A VARIABLE WHICH CONTAINS THE SUBROUTINE ENTRY ADDRESS FOR USE WITH THE JMPM INSTRUCTION. THE RETURN ADDRESS WORD FOR THE JMPM INSTRUCTION IS INITIALIZED TO ZERO. THE SEQUENCE "SUBROUTINE <NAME> -1 DP+!" IS USED WHEN THE SUBROUTINE IS TO BE CALLED WITH THE JSR INSTRUCTION. IT IS VERY IMPORTANT TO REMEMBER THAT FORTH'S COMPILATION FLAG IS NOT SET WHILE ASSEMBLING MACHINE CODE INSTRUCTIONS, THAT IS, FORTH REMAINS IN EXECUTION MODE. SEE RETURN.
- SWAP                  <VALUE1> <VALUE2> SWAP <VALUE2> <VALUE1>  
EXCHANGE THE TOP TWO VALUES ON THE STACK SO THAT <VALUE1> WILL BE THE TOP VALUE ON THE STACK AND <VALUE2> THE SECOND.

- T-H  
LOAD THE TAPE HANDLERS. SEE UTIL.
- T2D                   <BLOCK#> T2D  
READ THE SPECIFIED BLOCK FROM TAPE TO DISK. THE TAPE MAP MUST BE CONSTRUCTED BEFORE EXECUTING THIS WORD (SEE READ-MAP). SEE T-H.
- TAB,                (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TAB INSTRUCTION (TRANSFER THE A REGISTER TO THE B REGISTER).
- TAPE  
SETS THE TAPE AS THE PRIMARY DEVICE. SEE T-H.
- TAPE-TO-DISK        <START-BLOCK#> <END-BLOCK#> TAPE-TO-DISK  
FIRST ZERO ALL DISC BLOCKS IN THE SPECIFIED RANGE AND THEN COPY ALL BLOCKS IN THE RANGE FROM TAPE TO DISK. SEE T-H.
- TAX,                (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TAX INSTRUCTION (TRANSFER THE A REGISTER TO THE X REGISTER).
- TBA,                (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TBA INSTRUCTION (TRANSFER THE B REGISTER TO THE A REGISTER).
- TCH                   <CHAR-CODE> TCH  
TRANSMIT THE SPECIFIED ASCII CHARACTER CODE TO THE TERMINAL, REGARDLESS WHAT THE CURRENT OUTPUT DEVICE IS. SEE APPENDIX A FOR A LISTING OF THE ASCII CODES. SEE WCH.
- TEMP  
A VARIABLE USED FOR TEMPORARY STORAGE IN THE FLOATING-POINT ROUTINES.
- TERMINAL  
SELECT THE TERMINAL AS THE OUTPUT DEVICE, CANCELLING ANY PREVIOUS SELECTION OF THE PRINTER. THE TERMINAL IS AUTOMATICALLY SELECTED WHEN CONTROL IS RETURNED TO IT AFTER EXECUTING A LINE OF WORDS. SEE PRINTER.
- TERMINAL-ASK  
SETS AN INTERNAL FLAG SO THAT THE NEXT USE OF THE ASKING WORDS (SASK, DASK, FASK) WILL EXTRACT CHARACTERS FROM THE TERMINAL BUFFER. SEE BLOCK-ASK.
- TERMINAL-WORD (OLD)  
RENAMED TERMINAL-ASK.

TEX  
LOADS THE TERMINAL EXCHANGE WORDS WHICH ALLOW ONE TO SWITCH BETWEEN DIFFERENT TERMINAL DEVICES. SEE UTIL.

THEN (CO-,P)  
TERMINATE AN IF-ELSE-THEN CONDITIONAL. SEE IF.

THEN, (A)  
TERMINATE AN IF,-ELSE,-THEN, CONDITIONAL. SEE IF,.

TPABT  
FORCES A "?Y" ABORT.

TPLT <X-POSN> <Y-POXN> TPLT  
DRAWS A VECTOR ON THE 4010 FROM THE CURRENT POSITION TO THE SPECIFIED OFFSET FROM THE CURRENT POSITION AS GIVEN BY <X-POSN> AND <Y-POXN>. A DARK VECTOR IS THEN DRAWN BACK TO THE ORIGINAL POSITION. THE PARAMETERS TO THIS WORD ARE OFFSETS FROM THE CURRENT POSITION AND NOT PHYSICAL COORDINATES. <X-POSN> AND <Y-POXN> ARE SINGLE-WORD INTEGERS IN THE RANGE 0-1023 AND 0-780.

TSA, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TSA INSTRUCTION (TRANSFER THE FRONT PANEL SWITCH SETTING TO THE A REGISTER).

TWORD (OLD)  
REQUEST A LINE OF INPUT FROM THE TERMINAL. THE LINE IS TERMINATED BY A CARRIAGE RETURN AND THE FIRST WORD OF THE LINE IS THEN EXTRACTED AND PLACED IN MEMORY AT THE NEXT AVAILABLE DICTIONARY LOCATION.

TXA, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TXA INSTRUCTION (TRANSFER THE X REGISTER TO THE A REGISTER).

TYPE <COUNT> TYPE  
OUTPUT A STRING OF CHARACTERS TO THE OPERATOR'S TERMINAL. IP MUST CONTAIN THE BYTE-ADDRESS OF THE FIRST CHARACTER OF THE STRING AND <COUNT> SPECIFIES THE NUMBER OF CHARACTERS TO OUTPUT. SEE WRITE AND COUNT.

TZA, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TZA INSTRUCTION (TRANSFER ZERO TO THE A REGISTER).

TZB, (A)  
THE ASSEMBLER MNEMONIC FOR THE VARIAN TZB INSTRUCTION (TRANSFER ZERO TO THE B REGISTER).

- U.                   <VALUE> U.  
UNSIGNED OUTPUT. CONVERT <VALUE> ACCORDING TO THE CURRENT  
NUMBER BASE AND OUTPUT IT TO THE CURRENT OUTPUT DEVICE (USUALLY  
THE OPERATOR'S TERMINAL) AS A POSITIVE, UNSIGNED, 16-BIT  
NUMBER. THE VARIABLE FLD SPECIFIES THE FIELD WIDTH.
- UNPK                <DW-VALUE> UNPK <DW-RESULT>  
<DW-VALUE> IS ASSUMED TO BE A PACKED CAMAC 24-BIT VALUE WHICH  
IS UNPACKED TO YIELD A VARIAN DOUBLE-WORD INTEGER. SEE PACK.
- UNSAVER, (A)  
A WORD WHICH WILL RESTORE THE SYSTEM STATUS THAT WAS SAVED BY  
THE SUBROUTINE SAVER. THIS WORD MUST BE FOLLOWED BY THE MACHINE  
CODE REQUIRED TO RESTORE THE B REGISTER AND RETURN FROM AN  
INTERRUPT.
- UPDATE  
FLAG THE MOST RECENTLY REFERENCED BLOCK AS UPDATED. THE BLOCK  
WILL SUBSEQUENTLY BE TRANSFERRED AUTOMATICALLY TO DISC OR TAPE  
SHOULD ITS BUFFER BE REQUIRED FOR THE STORAGE OF A DIFFERENT  
BLOCK. SEE FLUSH.
- US  
PUTS THE 4010 IN ALPHA MODE. SEE GS.
- USER  
LOAD ADDITIONAL WORDS INTO THE DICTIONARY, NOTABLY THE  
DOUBLE-WORD INTEGER WORDS, THE FLOATING-POINT WORDS AND THE  
CAMAC WORDS.
- UTIL  
DEFINES THE FOLLOWING LOADER WORDS (REFER TO THE DESCRIPTION OF  
EACH WORD FOR FURTHER INFORMATION):  
    BRACKET, D-H, DISKO, FORMATTER, NEWTAPE, PRINTERS,  
    REDEF, T-H, TEX, ZERODISK.
- UTILITIES (OLD)  
    RENAMED UTIL.

- VARIABLE**                   <VALUE> VARIABLE <NAME>  
 DEFINE A WORD <NAME> WHICH, WHEN EXECUTED, WILL PUSH THE ADDRESS OF THE VARIABLE'S VALUE ONTO THE STACK. THE VALUE OF THE VARIABLE IS INITIALIZED TO <VALUE>. THE SEQUENCE "<NAME> @" PUSHES THE VARIABLE'S CURRENT VALUE ONTO THE STACK AND THE SEQUENCE "<VALUE> <NAME> !" STORES A NEW VALUE IN THE VARIABLE.
- VCHECK**  
 PRINTS SOME RELEVANT INFORMATION CONCERNING ALL VOCABULARIES (WHAT WORD IS THE HEAD OF THE VOCABULARY, WHICH VOCABULARIES ARE CHAINED TO OTHERS, ETC.).
- VLIST**  
 START LISTING THE DICTIONARY, BEGINNING AT THE HEAD OF THE CONTEXT VOCABULARY. THE LISTING MAY BE STOPPED BY PRESSING ANY TERMINAL KEY.
- VOCABULARY (E)**       VOCABULARY <NAME>  
 DEFINE A VOCABULARY WITH THE SPECIFIED NAME. SUBSEQUENT EXECUTION OF <NAME> WILL MAKE <NAME> THE CONTEXT VOCABULARY. THE SEQUENCE "<NAME> DEFINITIONS" WILL MAKE <NAME> THE CURRENT VOCABULARY, INTO WHICH DEFINITIONS WILL BE PLACED.
- W.D**                   <FIELD-WIDTH> <#DIGITS> W.D  
 SPECIFY BOTH THE TOTAL FIELD WIDTH AND NUMBER OF DIGITS TO THE RIGHT OF THE DECIMAL POINT FOR PRINTING FLOATING-POINT NUMBERS USING THE WORDS E. AND F.. THESE VALUES ARE STORED INDEPENDENTLY OF THE VARIABLES FLD AND DPL, HOWEVER, EACH TIME EITHER E. OR F. IS EXECUTED THE VALUES OF FLD AND DPL WILL CHANGE. STORING NEW VALUES IN FLD AND DPL DOES NOT AFFECT THE FLOATING-POINT FIELD SPECIFICATIONS. THE DEFAULT VALUES ARE A FIELD WIDTH OF 14 AND 4 DIGITS TO THE RIGHT OF THE DECIMAL POINT.
- WAIT**               (A)  
 THE ADDRESS OF A SUBROUTINE WHICH SHOULD BE USED WHENEVER A PROGRAM MUST WAIT FOR ANY FORM OF I/O. USER SWAPPING IS DONE IN WAIT. THIS SUBROUTINE MUST BE CALLED BY A JMPM INSTRUCTION.
- WCH**               <CHAR-CODE> WCH  
 OUTPUT THE SPECIFIED ASCII CHARACTER CODE TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). SEE APPENDIX A FOR A LISTING OF THE ASCII CODES. SEE TCH.

WF

INITIATES THE WRITING OF AN END-OF-FILE ON THE MAG TAPE AND RETURNS IMMEDIATELY. SEE ENDFILE.

WG

INITIATES THE WRITING OF A THREE INCH GAP ON THE MAG TAPE (REFERRED TO AS ERASING THE TAPE) AND RETURNS IMMEDIATELY. SEE WGAP.

WGAP

INITIATE THE WRITING OF A THREE INCH GAP ON THE MAG TAPE (REFERRED TO AS ERASING) AND WAIT FOR THE OPERATION TO COMPLETE. SEE WG.

WHERE

OUTPUT INFORMATION ABOUT THE STATUS OF FORTH. THIS WORD IS USUALLY EXECUTED AFTER AN ERROR ABORT TO DETERMINE WHERE THE SYSTEM WAS (LAST WORD COMPILED AND LAST BLOCK ACCESSED) WHEN THE ERROR OCCURED.

WHILE

(C2+,P) <LOGICAL-VALUE> WHILE  
TEST THE <LOGICAL-VALUE> AND IF FALSE, EXIT OUT OF A BEGIN-WHILE-REPEAT LOOP IMMEDIATELY. SEE BEGIN.

WINIT

INITIALIZE THE WORD REPLACEMENT ARRAY. THIS WORD MUST BE EXECUTED PRIOR TO A SERIES OF REPLACE AND/OR \$REPLACE. SEE FIX, REPLACE AND \$REPLACE.

WORD

<CHAR-CODE> WORD  
READ THE NEXT WORD FROM THE INPUT STRING BEING INTERPRETED. THE WORD IS TERMINATED BY THE SPECIFIED <CHAR-CODE> AS THE DELIMITER (FORTH USUALLY SPECIFIES A BLANK AS A DELIMITER, ALTHOUGH ANY ASCII CHARACTER MAY BE SPECIFIED). THE PACKED CHARACTER STRING IS STORED IN THE DICTIONARY, BEGINNING AT THE NEXT AVAILABLE LOCATION (SEE THE WORD HERE) WITH THE CHARACTER COUNT IN THE FIRST BYTE.

WORDIN

WORDIN <ADDRESS>  
REQUEST A WORD FROM THE TERMINAL. A STRING OF CHARACTERS TERMINATED BY A CARRIAGE RETURN IS READ IN FROM THE TERMINAL AND THE FIRST WORD (ALL CHARACTERS UP TO THE FIRST BLANK) IS STORED IN THE NEXT AVAILABLE DICTIONARY LOCATION AND ITS ADDRESS IS RETURNED ON THE STACK.

WRITE

<COUNT> WRITE  
OUTPUT A STRING OF CHARACTERS TO THE CURRENT OUTPUT DEVICE (USUALLY THE OPERATOR'S TERMINAL). IP MUST CONTAIN THE BYTE-ADDRESS OF THE FIRST CHARACTER OF THE STRING AND <COUNT> SPECIFIES THE NUMBER OF CHARACTERS TO OUTPUT. THIS WORD DIFFERS FROM TYPE IN THAT WRITE WILL OUTPUT TO THE CURRENT OUTPUT DEVICE (WHICH MAY OR MAY NOT BE THE OPERATOR'S TERMINAL) WHILE TYPE WILL OUTPUT ONLY TO THE OPERATOR'S TERMINAL. SEE COUNT.



- X) (A)  
SETS THE VARIABLE MODE TO 5, SPECIFYING INDEXING OFF THE X REGISTER FOR THE NEXT MEMORY REFERENCE INSTRUCTION. NOTE THAT FORTH KEEPS THE CURRENT STACK POINTER IN THE X REGISTER.
- XIF, (A) <ADDRESS> <JUMP-CONDITION> XIF,  
THE ASSEMBLER MNEMONIC FOR THE VARIAN XIF INSTRUCTION (EXECUTE THE INSTRUCTION AT <ADDRESS> IF THE CONDITION IS TRUE).
- XOR <VALUE1> <VALUE2> XOR <RESULT>  
FORM THE BITWISE LOGICAL EXCLUSIVE-OR OF <VALUE1> AND <VALUE2>, LEAVING THE RESULT ON THE STACK.
- XY4010 <X-POSN> <Y-POSN> XY4010  
DRAWS A VECTOR ON THE 4010 TO THE PHYSICAL LOCATION SPECIFIED BY <X-POSN> AND <Y-POSN>. <X-POSN> AND <Y-POSN> ARE SINGLE-WORD INTEGERS IN THE RANGE 0-1023 AND 0-780.
- ZAP  
DELETE THE ENTIRE DICTIONARY, RESET ALL INTERNAL POINTERS AND FLAGS, RELOAD BASIC FORTH (BLOCK 8) INTO THE DICTIONARY.
- ZERODISK <START-BLOCK#> <END-BLOCK#> ZERODISK  
ZEROS THE SPECIFIED RANGE OF BLOCKS ON THE DISK. SEE UTIL. SEE DISKO.
- [ (P)  
STOP COMPILATION. THE WORDS FOLLOWING THE LEFT BRACKET IN A COLON DEFINITION ARE EXECUTED, NOT COMPILED. SEE ].
- [P]  
SET THE PRECEDENT BIT OF THE NEXT WORD DEFINED SO THAT THE WORD IS A COMPILER DIRECTIVE.
- ]   
RESUME COMPILATION. THE FOLLOWING WORDS IN A COLON DEFINITION ARE COMPILED, NOT EXECUTED. SEE [.

## SINGLE-WORD INTEGER ARITHMETIC WORDS

-----

\*  
\*\*  
\*/  
+  
-  
/  
/MOD  
ABS  
MAX  
MIN  
MINUS  
MOD

## DOUBLE-WORD INTEGER ARITHMETIC WORDS

-----

ASHIFT  
D\*/  
D+  
D-  
DABS  
DMINUS  
SORT

## MIXED-PRECISION INTEGER ARITHMETIC WORDS

-----

2M\*  
M\*  
M+  
M/  
M/MOD

## FLOATING-POINT ARITHMETIC WORDS

-----

\*10\*\*  
 F\*  
 F+  
 F-  
 F/  
 F2LOG  
 F2XP  
 FABS  
 FDATN  
 FDCOS  
 FDSIN  
 FDTAN  
 FEXP  
 FEXP10  
 FLN  
 FLOG  
 FMAX  
 FMIN  
 FMINUS  
 FRATN  
 FSQRT

## 14-BIT FRACTION ARITHMETIC WORDS

-----

\*,  
 /,  
 R1-2  
 RS.C  
 SIN.COS

## DOUBLE-WORD FRACTION ARITHMETIC WORDS

-----

D\*  
 D/

## NUMERIC CONVERSION

-----

|        |                |    |                |
|--------|----------------|----|----------------|
| .FIX   | FLOATING-POINT | -> | DW-FRACTION    |
| .FLOAT | DW-FRACTION    | -> | FLOATING-POINT |
| DFIX   | FLOATING-POINT | -> | DW-INTEGER     |
| DFLOAT | DW-INTEGER     | -> | FLOATING-POINT |
| SFIX   | FLOATING-POINT | -> | SW-INTEGER     |
| SFLOAT | SW-INTEGER     | -> | FLOATING-POINT |

## SINGLE-WORD LOGICAL OPERATORS

-----

|       |                 |
|-------|-----------------|
| AND   |                 |
| COM   | ONES COMPLEMENT |
| MINUS | TWOS COMPLEMENT |
| OR    | INCLUSIVE-OR    |
| XOR   | EXCLUSIVE-OR    |

## SINGLE-WORD STACK OPEARATORS

-----

DROP  
DUP  
LS  
NDROP  
OVER  
PICK  
ROLL  
ROT  
SWAP

## 2-WORD STACK OPEARATORS

-----

2DROP  
2DUP  
2LS  
2OVER  
2PICK  
2ROLL  
2ROT  
2SWAP

## 3-WORD STACK OPERATORS

-----

3DROP  
3DUP  
3OVER  
3PICK  
3ROLL  
3ROT  
3SWAP

## NUMERIC OUTPUT WORDS

-----

|    |                                  |
|----|----------------------------------|
| .  | SINGLE-WORD INTEGER, FREE FORMAT |
| ?  | SINGLE-WORD INTEGER, FREE FORMAT |
| B. | SINGLE-WORD INTEGER, BINARY      |
| D. | DOUBLE-WORD INTEGER              |
| E. | FLOATING-POINT WITH EXPONENT     |
| E? | FLOATING-POINT WITH EXPONENT     |
| F. | FLOATING-POINT WITHOUT EXPONENT  |
| F? | FLOATING-POINT WITHOUT EXPONENT  |
| G. | GENERALIZED FLOATING-POINT       |
| G? | GENERALIZED FLOATING-POINT       |
| H. | SINGLE-WORD INTEGER, HEXADECIMAL |
| I. | SINGLE-WORD INTEGER              |
| N. | SINGLE-WORD INTEGER              |
| O. | SINGLE-WORD INTEGER, OCTAL       |
| S. | SINGLE-WORD INTEGER              |
| U. | SINGLE-WORD INTEGER, UNSIGNED    |

## COLON DEFINITION CONTROL

-----

+LOOP  
BEGIN  
CASE  
DO  
ELSE  
END  
IF  
I  
I'  
J  
K  
LEAVE  
LOOP  
REPEAT  
THEN  
WHILE

## MACHINE CODE CONTROL

-----

BEGIN,  
ELSE,  
END,  
IF,  
THEN,

## DIRECT MAG-TAPE WORDS (7-TRACK)

-----

|          |                                 |
|----------|---------------------------------|
| >BCD     | FLAG FOR BCD OR BINARY MODE     |
| ?EOF     | TEST FOR END-OF-FILE            |
| ?MTREADY | TEST FOR READY                  |
| BK       | INITIATE BACK SPACE RECORD      |
| BKSP     | BACK SPACE RECORD AND WAIT      |
| ENDFILE  | WRITE AN EOF AND WAIT           |
| FW       | INITIATE A FORWARD SPACE RECORD |
| FWSP     | FORWARD SPACE RECORD AND WAIT   |
| LWA?     | READ IN FINAL DMA ADDRESS       |
| MTPERR   | TEST FOR PARITY ERROR           |
| MTR      | INITIATE A READ                 |
| MTREAD   | READ AND WAIT                   |
| MTREJ    | TEST FOR REJECT                 |
| MTW      | INITIATE A WRITE                |
| MTWAIT   | WAIT FOR READY                  |
| MTWRITE  | WRITE AND WAIT                  |
| OFFLINE  | TURN THE DRIVE OFFLINE          |
| REW      | REWIND AND WAIT                 |
| RW       | INITIATE A REWIND               |
| SKIPF    | SKIP ONE FILE AND WAIT          |
| TPABT    | FORCE A "?Y" ABORT              |
| WF       | INITIATE THE WRITING OF AN EOF  |
| WG       | INITIATE THE WRITING OF A GAP   |
| WGAP     | WRITE A GAP AND WAIT            |