OmniFORTH

INTERACTIVE COMPUTER SYSTEMS, INC.
6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

First Edition 1979

1 OmniFORTH INTRODUCTION

2 OmniFORTH EDITOR

OmniFORTH

Introduction

INTERACTIVE COMPUTER SYSTEMS, INC.

6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

1.1 Introduction

This text is not intended to be an all inclusive text but it is intended to provide elementry OmniFORTH concepts in a simple easy-to-master manner.

A small subset of OmniFORTH words have been selected that will teach the OmniFORTH fundamentals without overpowering the reader.

It is hoped that once this text is understood the reader will be able progress quickly on his own in mastering the complete set of words listed in the glossary.

1.1.1 What is OmniFORTH?

OmniFORTH is more than just a computer language, it is a language, an operating system, an editor, a monitor and assembler all in one extensible package.

OmniFORTH is like a spoken language in that it is a collection of defined words which are verbs, nouns and modifiers. OmniFORTH is used, not in writing programs, but defining new words and then using the words to produce the desired results. Therefore, OmniFORTH is simply a dictionary of words that includes all defined words whether they are new words defined by the user or old words previously defined and supplied in the OmniFORTH package.

OmniFORTH can be thought of as 16 bit stack orientated computer language that combines structured programming, virtual memory, compiler, assembler, and file system into an efficient extensible macro-language.

1.1.2 OmniFORTH is Interactive

Just as English is interactive (a person says RUN and the person spoken to runs) the OmniFORTH user may enter the word LIST and the specified text is listed on the console.

Each new word entered is ready for execution after it is defined. This means that you can create a new word, test it immediately, and debug it as you go using the interactive features of OmniFORTH. The interactiveness of OmniFORTH will enable your software development effort time to be reduced to a fraction of that required by other languages.

1.1.3 OmniFORTH is Structured

Just as English is structured, OmniFORTH has no GO TO since, when speaking, it would be impractical to say:

"GO TO THE WORD I SPOKE FIVE MINUTES AGO"

It is impractical in OmniFORTH to GO TO a word that was executed previously. OmniFORTH is also modular like English because once a word is defined it is no longer necessary to repeat the definition to get the desired response.

1.1.4 OmniFORTH is Extensible

Extensibility is the ability to define new words (as previously mentioned) that customize a language to say or do what we wish. New words are added by the user into the OmniFORTH dictionary until the user has developed a vocabulary that suits his needs.

1.1.5 Present Day OmniFORTH

OmniFORTH is still in its infancy. The FORTH language is becoming popular as a control language and as a man-machine and machine-machine interface. More serious users are turning to FORTH because it gives them complete control of their computer and it is cost effective. Applications can be written in OmniFORTH that require less harware and software investment than other high level languages.

1.2 The Stack and Arithmetic

This section deals with console operation and the stack and arithmetic operators.

OmniFORTH is a reverse-polish language: this section may make OmniFORTH look like a reverse-polish calculator, but later you will find that reverse-polish is the natural order of computers.

The stack is illustrated with the top to the right. For example:

Sn . . . S3 S2 S1

is the notation showing S1 as top of the stack, S2 as the second stack item, S3 as the third item on the stack, and so forth until Sn which is the nth item of the stack. Since S1 is the top of the stack, S2 is below S1, S3 is below S2, and Sn is below Sn-1.

1.2.1 Stack Operators

The following is a list of the operators used in this introduction (additional operators are given in glossary).

Word	Explanation	Stack Before	Stack After
DROP	Removes the top stack item.	4 5	4
DUP	Duplicate the top stack item.	2	2 2
OVER	Copy second stack item over first, placing it on top.	2 1	2 1 2
ROT	Rotate the top three stack items, bringing the third to the top of the stack.	1 3 2	3 2 1
SWAP	Swap the top two stack items.	6 1	1 6

1.2.2 Arithimetic Operators

Unless otherwise noted, all numbers are assumed 16 bit signed integers. All arithemetic is implicitly 16 bit signed integer math. Illustrations assume that the normal base is decimal and any other base will be noted on examples.

The following is a list of arithmetic operators used in this introduction (more are given in glossary).

Word	Explanation	Stack Before	Stack After
+	Add the two top stack values leaving the sum.	2 4	6
-	Subtract top stack value from the second, leaving the difference.	5 2	3
*	Multiply the two top stack values, leaving a signed 16 bit integer number.	4 2	8
/	Divide second stack value by the top stack item, leaving a signed 16 bit integer number.	7 2	3
/MOD	Divide second stack value by the top stack item, leaving the quotient on top and signed reminder beneath.	7 2	1 3

1.2.3 Output Operator

The output operator "." pronounced "dot" will convert a 16 bit signed number found on top of the stack and then print it on the console with a trailing space. Conversion is done using the current numeric base.

Word	Explanation	Stack Before	Stack After
.	Display the top stack value as a signed number, dropping the value from the stack.	3	(type "3" dropping value)

1.3 Notations and Examples

For clarity and brevity, we use some special notations to illustrate the OmniFORTH language in the following text.

User entries will be underlined and OmniFORTH output will not.

The symbol (CR) will mean that the user enters a carriage return.

1.3.1 Example: Enter a Number on the Stack

The user enters a number by typing it on console and terminating it with a carriage return.

```
53 (CR)  OK
```

Explanation: OmniFORTH scans the input buffer until the string 53 delimited by space or carriage return is found. The string is tested against defined words in the vocabulary and finally is converted to a 16 bit signed integer number (base 10, decimal in this example) and placed on top of the stack. OmniFORTH then displays OK as a prompt indicating the request is complete and it is now ready for more input.

1.3.2 Example: Enter Number and Print

User enters another number and attempts to print S1, S2, and S3.

```
78  .  .  .  (CR)  78  53  XX .? EMPTY STACK
```

Explanation: 78 was placed on top of the stack and 53 is below it at S2 (the number 53 was entered in the previous example). The first period causes S1 to print 78 and drop it from the top allowing S2 to become the new S1 top stack item. The second period now prints 53 and drops it from the top thus emptying the stack. Finally the third period attempts to print an empty stack and the error message "XX .? EMPTY STACK" is typed by OmniFORTH since there was nothing left on the stack. Note that in generating the error message the "XX" represents the attempted output of 53 and the ".? EMPTY STACK" was displayed as the error on "." trying to type the empty stack.

For clarity and brevity, the example above can be illustrated using the following notation:

Execution	Stack (top of stack is to the right)
example 1	53 (example 1 left 53 on stack)
78	53 78 (put 78 on top of stack over 53)
.	53 (type "78" leaving 53 on top)
.	(type "53" leaving stack empty)
.	(type ".XX STACK EMPTY" error message)
(CR)	end of input buffer

1.3.3 Example: Addition and Print

User adds two numbers together and prints the sum.

```
5  2  +  .  (CR)  7  OK
```

Explanation of the stack at each operation:

Execution	Stack (top of stack is to the right)
5	5
2	5 2
+	7
.	(type "7" leaving stack empty)
(CR)	end of input buffer

1.3.4 Example: Tax Calculation

User wishes to calculate 5% of 2900 and print value.

```
2900  5  *  100  /  .  (CR)  145  OK
```

Explanation of the stack at each operation:

Execution	Stack (top of stack is to the right)
2900	2900
5	2900 5
*	14500
100	14500 100
/	145
.	(type "145" leaving stack empty)
(CR)	end of input buffer

1.3.5 Example: All the Operators

This example will illustrate all the operations listed in the beginning of this section.

```
4  3  2  1  (CR)
SWAP  OVER  DROP  ROT  DUP  +  *  (CR)
OVER  -  ROT  /MOD  /  .  .  (CR)  1  1  OK
```

Explanation of the stack at each operation:

Execution	Stack (top of stack is to the right)
4	4
3	4 3
2	4 3 2
1	4 3 2 1
(CR)	end of first line
SWAP	4 3 1 2
OVER	4 3 1 2 1
DROP	4 3 1 2
ROT	4 1 2 3
DUP	4 1 2 3 3
+	4 1 2 6
*	4 1 12
(CR)	end of second line
OVER	4 1 12 1
-	4 1 11
ROT	1 11 4
/MOD	1 3 2
/	1 1
.	1 (type "1" leaving 1 on top of stack)
.	(type "1" leaving stack empty)
(CR)	end of third line

1.4 Defining New Words (Compiling)

In the previous section we illustrated the stack and several predefined words. We are now going to show how you can extend the OmniFORTH dictionary by defining new words in terms of existing words and numbers.

Extensibility of OmniFORTH will enable you to define your own words to best describe your application and instruct the computer in your own terms.

Words are definded as a string of one or more characters delimited by at least one space or a carriage return.

OmniFORTH will search for the word in the dictionary; if it is found it will be executed.

If the word is not found in the dictionary, OmniFORTH will attempt to convert the word to a number using the current number base.

If the word (character string) is successfully converted to a signed integer number, the number is placed on top of the stack.

If all fails, (the string was not a defined word or it could not be converted to a number) OmniFORTH will type the character string, print an error message, clear the stack, and finally stop interpretation of the input buffer.

1.4.1 The Colon Definition

At this time, we introduce the colon ":" definition. The colon is a word that is a part of OmniFORTH and enables you to define new words.

In the last section we demonstrated that the user could enter and execute the following interactively:

```
2900  5  *  100  /  . (CR)  145  OK
```

State sales tax may be 5% and the user may wish to define a new word TAX that will compute and print the sales tax for a variety of items. The user would enter:

```
:  TAX  5  *  100  /  .  ;  (CR)  OK
```

The colon ":" is a word that has been defined to compile into the OmniFORTH dictionary a definition header containing the next word appearing in the input buffer "TAX". TAX will then become a new word definition that will cause the body "5 * 100 / ." to be executed whenever TAX is used. The ";" is another predefined OmniFORTH word that signals the end of a colon ":" definition. A new word is structured as follows:

```
:  TAX  5  *  100  /  .  ;
↑  ↑    ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑  ↑
!  !    !                end of definition
!  !    !
!  !    body of definition
!  !
!  name of definition
!
compiler directive
```

Note that the colon ":" also places OmniFORTH in compiler mode and directs that the following words in the body of the definition to be compiled in the dictionary instead of being executed. The compiling will continue until the semicolon ";" signals the end of a definition. Remember that OmniFORTH expects a space or carriage return to delimit words, so be sure to preceed and follow each word with a space and end the input string with a carriage return.

We now have a new word TAX defined in our vocabulary that will compute 5% of an item and print it. Therefore we may now enter:

```
      2900  TAX  (CR)  145  OK
or    2000  TAX  (CR)  100  OK
or     100  TAX  (CR)    5  OK
```

1.5 Conditionals

No language would be complete without the ability to make conditional branches and loops. This section will introduce the IF ... ELSE ... ENDIF, DO ... LOOP, DO ... +LOOP, and the BEGIN ... UNTIL constructions. It should be noted that other conditionals are available but will not be covered in this text, they are left to the reader to investigate.

Conditionals cannot be used outside of colon definitions that define a word. Conditionals are made up of special words such as the IF ... ELSE ... ENDIF that depend on each other and on the word being defined for conditional execution.

All OmniFORTH conditionals use a condition value placed on top of the stack to make a decision. A zero on top of the stack sets the false condition (0=FALSE). The true condition is set when the top of the stack is not a zero. The top of stack (true or false) is removed by the condtional. If a condition is needed later, the condition must be duplicated or saved for later reference.

1.5.1 Comparison Words

In order to provide examples of IF and UNTIL conditional operators, the following comparison words are listed:

Word	Explanation	Stack Before	Word	Stack After
0=	Leave a true if top of stack was equal to zero	0	0=	1
		5	0=	0
0<	Leave a true if top of stack was less than zero	-4	0<	1
		4	0<	0
=	Leave a true if top two stack items were equal	7 7	=	1
		1 3	=	0
<	Leave a true if item under top stack is less than top	2 6	<	1
		9 4	<	0
>	Leave a true if item under top stack is greater than top	6 2	>	1
		4 9	>	0

1.5.2 IF Statement

The IF statement must occur within a colon definition and has the following forms:

```
f  IF  (tp)...  ENDIF
```

or

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

Where:

f — is a Boolean value on top of the stack. 0 = FALSE, non-zero = TRUE

(tp) — is the true part that executes if f is true.

(fp) — is the false part that executes if f is false.

Explanation:

The IF selects execution based on a Boolean flag f that preceeds the IF. The Boolean flag is removed from the stack when the IF is executed.

When f is true (non-zero), execution continues thru the true part (tp) and skips over the ELSE and false part (fp), if used, to just after ENDIF.

When f is false (zero), execution skips over the true part to just after ELSE and executes the false part (fp), if present, and continues thru the ENDIF.

1.5.3 Example: IF (tp)... ELSE (fp)... ENDIF

The following example will print out "Yes" if the top of the stack S1 is true and "no" if S1 is false:

```
: YES/NO IF ." YES"  ELSE  ." NO"  ENDIF ;
```

This routine will remove the top stack item S1 and execute ." YES" if S1 was true (not equal zero) otherwise the ." NO" will be executed. The routine can be tested at the console as:

```
0  YES/NO  (CR)   NO   OK
1  YES/NO  (CR)   YES  OK
```

1.5.4 Example: IF (tp)... ENDIF

Leave the smallest of the top two stack items.

```
: MIN  OVER  OVER  >  IF  SWAP  ENDIF  DROP  ;  (CR)
```

Explanation of how the routine works:

```
OVER OVER        ( Duplicates top two stack items)
>                ( leave true  if S2 > S1)
IF SWAP ENDIF    ( Swap will execute if S2 > S1)
DROP             ( Drops the top (largest) stack item
                   leaving minimum on the stack)
```

1.5.5 Nesting IF Statements

IF statements may be nested many times. The limit of nesting may vary between implementations, but in most cases the user will run out of need before the limit will be reached. The following example is not a usable operation, but illustrates the nesting capability.

```
: NEST-IF
          f IF (tp)...
                    f IF (tp)...
                              f IF (tp)...
                                        ELSE (fp)...
                              ENDIF
                    ELSE (fp)...
          ENDIF
ELSE (fp)...
ENDIF ;
```

1.5.6 DO ... LOOPs

DO loops provide the capability of looping a known number of times. DO loops are contructed in two forms:

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

and

```
n1 n2 DO ... n3 +LOOP
```

where:

n1 is the loop limit or final value

n2 is the initial index value

n3 is the loop increment
(note LOOP is equivalent to 1 +LOOP)

At execution time, the DO sets the loop index I to the initial value n2 and saves the loop limit n1. Both n1 and n2 are removed from the parameter stack. Then a sequence of repetitive executions begins controlled by the loop limit n1. Upon reaching the LOOP, or +LOOP the index I is incremented by 1, or n3 and then tested. If the index is less than the loop limit n1, execution loops back to just after the DO. When the index I finally increments to be equal or greater than the loop limit, the loop terminates removing n1 and n3 from the stack.

Note that the word "I" may be used within a DO ... LOOP to place the loop index on the stack.

1.5.7 Examples: DO ... LOOP

The following are two examples that will illustrate the operation of each type of DO ... LOOP:

```
: ABC 10 0 DO I . LOOP ; (CR)  OK

ABC (CR)   0 1 2 3 4 5 6 7 8 9  OK

: XYZ 10 0 DO I . 2 +LOOP ; (CR)  OK

XYZ (CR)   0 2 4 6 8  OK
```

It should be noted that the numbers n1, n2, and n3 are removed from the parameter stack and n1 and n2 are placed on the return stack by the DO so that they are not directly available within the DO ... LOOP.

1.5.8 Example: Memory Dump Using DO ... LOOP

A simple memory dump routine may be constructed using a single DO ... LOOP. In this case we wish to enter the first memory address, last memory address and the word DUMP.

```
: DUMP  1+ SWAP DO I C@ . LOOP ; (CR) OK
```

To use the DUMP enter:

```
HEX  1000  1005  DUMP  (CR)  75  5  AF  B7  1  9  OK
```

In this example we altered the revrse polish parameters n1 and n2, since it seems more natural to enter the starting address before the ending address. To do this we execute the SWAP. To include the byte at the ending address we add 1 to the n1 before the SWAP. The loop index will now increment from 1000 to 1005. The I will place the loop index on the stack, the C@ will replace the top of the stack with memory contents pointed to by the loop index and the dot "." will print the memory contents using HEX base conversion.

WARNING: It is quite common for new OmniFORTH programmers to leave one or more items on the stack at each loop. If the number of loops are large, the stack may overflow and corrupt the memory resident OmniFORTH system. Use caution when testing loops by using small values and check the stack to see if you have left any items on it by accident.

1.5.9 Nesting DO ... LOOPs

The DO ... LOOP may be nested simialar to the IF such as:

```
: NEST-DO
        n1 n2 DO ...
                    n1 n2 DO ...
                                n1 n2 DO ...
                                LOOP
                    LOOP
        LOOP ;
```

Although the DO provides tremendous power, often the user needs a loop that is independent of a count, and will loop until some condition is true. The BEGIN ... UNTIL loop provides this capability.

1.5.10 BEGIN ... UNTIL loop.

The BEGIN .... UNTIL loop is constructed as follows:

```
BEGIN ... f UNTIL
```

Where BEGIN marks the start of a sequence that may be repetitively executed. BEGIN serves as a return point for the UNTIL. If the Boolean flag f is false (zero), the UNTIL causes execution to return to BEGIN. Only when the flag is true (non-zero), does the UNTIL force the loop to terminate and makes the execution skip to the word after UNTIL.

The following is an example that will accept input from the console, then echo the input to the console, and place the input character in a buffer. The input will be terminated by entry of a carriage return. At termination the counter CNT will contain the number of input characters entered by the user. It should be noted that this routine bypasses the normal OmniFORTH input.

```
DECIMAL                          ( set base 10 )

0 VARIABLE BUFFER 78 ALLOT       ( dimension Buffer(80))

0 VARIABLE CNT                   ( define CNT=0 )

: INPUT                          ( define word INPUT)
        BEGIN                    ( begin .....)
           KEY DUP               ( input and duplicate)
           EMIT                  ( echo character)
           DUP                   ( duplicate character)
           BUFFER                ( get BUFFER address)
           CNT @                 ( get value of CNT)
           +                     ( add to BUFFER address)
           C!                    ( store char at BUFFER(CNT))
           1 CNT +!              ( increment CNT)
           13 =                  ( test char DUPed before
                                       for carriage return)
        UNTIL                    ( exits when char = CR)
  10  EMIT  ;                    ( output line feed )
```

1.6 Reference Suppliment

This section is designed to be a reference suppliment to the previous sections. The disscussions are specific areas of the use of OmniFORTH and it is assumed that the reader now has some knowledge of OmniFORTH and is familiar with the use of the glossary.

1.6.1 Number Base Conversions

In theory OmniFORTH can use any number base for input and output. The conversion base is stored in a user variable called BASE.

The OmniFORTH initial load or COLD start will set BASE to 10 (DECIMAL). The user may change the conversion base to any value from 2 to 36 by entering:

```
n  BASE  !     ( where n is a number 2 to 36 )
```

Now all input and output numbers will be converted according to the current value stored in BASE. In actual practice most of the time a user will use either DECIMAL, OCTAL, or HEX numbers. OmniFORTH's vocabulary contains predefined words, DECIMAL, OCTAL, and HEX that set the BASE to 10, 8, or 16.

The following is an example of an interactive conversion of a DECIMAL number to HEX:

```
DECIMAL  255  HEX  .  (CR)  FF  OK
```

The above example sets the BASE to 10, places the number 255 on the stack, sets the BASE to 16, and then . converts and prints the number using BASE 16. Note that the BASE is now set to 16 and all input and output numbers will be in HEX.

1.6.2 Constants

In any computer language it is necessary to have different data types. OmniFORTH has a data constant declaration word CONSTANT that can be used to define words that will place a predetermined numeric value on top of the stack.

The following is an example of the use of CONSTANT:

```
HEX                         ( set BASE 16)
0  CONSTANT ZERO            ( define 0=ZERO)
1  CONSTANT ONE             ( define 1=ONE)
FF CONSTANT MASK            ( define FF=MASK)
DECIMAL                     ( set BASE 10)
```

The above example will create three words ZERO, ONE, and MASK that when executed will place a 0, 1, and 255 on the stack.

1.6.3 Variables, Arrays and Buffers

Variables differ from constants because when they are executed the address of the data is placed on the stack. Therefore to get at the data stored in a variable we must use a the word @ to fetch the contents of an address on top of the stack. Example:

```
0  VARIABLE  VALUE          ( initializes 0=VALUE)
23 VALUE !                  ( stores 23 at VALUE)
VALUE  @                    ( fetch value at VALUE)
```

Arrays and buffers are about the same and can be created using the words VARIABLE and ALLOT. An eighty character buffer or array can be created as:

```
0  VARIABLE  BUF  78  ALLOT
```

Note that VARIABLE defines a word, BUF in this example, that is 16 bits (two bytes) long and initializes it to zero. Next the example continues to allocate an additional 78 bytes using ALLOT to extend the BUF array to embrace a total of eighty bytes of memory. It should be noted that only the first two bytes have been initialized and the remaining 78 bytes are not.

Remember that all references to a variable will cause the a 16 bit address of the variable to be placed on the stack. Any byte in the buffer may be accessed by adding an offset value to the address. For example the fifth byte may be accessed by:

```
BUF 4 + C@
```

1.6.4 Double Numbers (32 Bit)

OmniFORTH has the ability to use 32 bit double integer numbers to allow computations requiring more precision than that available with 16 bit numbers.

Double numbers are constructed as 32 bit signed integers occupying two 16 bit stack positions or memory locations. The top of the stack holds the most significant part including sign, and the second stack item contains the least significant part of the double number.

This section will cover input and output of double numbers. The math operators available will not be discussed here since they are well described in the glossary (such as M*, M/, D+ etc.).

On input the interpreter will treat a number containing a decimal point (such as 1.000000) as a double number.

The number is converted as an integer. The decimal point will not effect the converted value therefore 1.000000 and 100000.0 will convert to the same 32 bit signed integer number. The only difference is the value placed in user variable DPL.

DPL is set to the number of digits to the right of the decimal point and it is up to the user to use DPL as needed.

Double numbers can be displayed using "D." to perform the conversion from 32 bit signed integer number into digits that can be printed on the console. For further custom formatting see the next section.

1.6.5 Custom Number Formatting

Custom formatting may seem complex at first and is not necessary for most applications.

The following WORDS are used in number formatting:

<#, #, #>, SIGN, HOLD, and #S

The glossary should be referred to for definitions and use of the operators.

In general <# setups for a double number conversion. # or #S are used to convert the digits and #> is used to terminate the conversion. The words SIGN and HOLD are used to insert the sign and or other characters such as decimal points, commas, etc.

The converted digits begin with the last significant digit being stored at PAD-1 and each succeeding digit placed below that at bytes PAD-2, PAD-3, etc.

#> places the count of characters and address of the converted string on the stack in the order required for the word TYPE which will output the string.

The above discussion is not complete and will require experimentation of the user. In hopes of helping, the coding of output operators we have included listings of screens that are supplied with OmniFORTH. The OmniFORTH INSTALLATION manual contains source listings for several custom number routines along with source for the OmniFORTH EDITOR that can be used for reference.

OmniFORTH

Editor

INTERACTIVE COMPUTER SYSTEMS, INC.

6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

2.1 OmniFORTH Text Editor

The OmniFORTH operating system contains an EDITOR vocabulary. The EDITOR allows you to build and maintain text on disk. New text and changes to existing text are made interactively using the EDITOR . Insertion, deletion, and replacement of text are allowed.

Some of the features of the EDITOR are:

Text may be listed.

New lines may be inserted.

Existing lines may be altered or deleted.

Lines may be copied or moved on a screen.

Text on one screen can be copied to another screen.

Character strings in the text may be found and changed.

Internal pointers to line and cursor are under your control.

Edit command definitions can be called by your application.

New commands can be added to existing set to extend EDITOR.

The source for the EDITOR is provided on the OmniFORTH disk starting at screen 36. We hope that you will use it to learn more about OmniFORTH and encourage you to extend it to suit your needs.

CAUTION

The EDITOR gives you complete control and access to all information stored on disk. We recommend that you make a backup copy of your disk before starting an edit session. Use your Disk Operating System DOS to copy the entire disk for backup.

You will be editing interactively and directly onto disk. Need for backup will be greatest while you are learning how to use the system and will diminish as you gain experience. Even experienced users will occasionally make mistakes and destroy valuable information on disk, so please use caution and take the time to make a backup copy of your disk.

2.2 EDITOR Commands

The EDITOR responds to the following command vocabulary shown alphabetically.

All commands are shown in upper case. Parameters and user supplied data are indicated in lower case. String indicates one or more characters, and numbers are indicated by n and m. Each command is terminated by typing a carriage return.

Use the command TOP to reset error flags and restart edit.

The cursor position is shown by the _ underline character.

The EDITOR uses OmniFORTH variables PAD (to allocate text buffers) and SCR (to indicate which screen is being edited).

Note (part of OmniFORTH *) indicates a command available outside of the EDITOR vocabulary. They are included here because they are useful when editing.

COMMAND	DESCRIPTION
B	Backup cursor by length of text in PAD.
C string	Copy string into line at cursor.
n CLEAR	Clear entire screen n to spaces.
n m COPY	Copy all text from screen n to screen m.
n D	Delete line n by holding line at PAD and moving lines n+1 thru 15 upwards. Line 15 will be reproduced. Note: 15 D won't work because n+1 is off screen. Use 15 H 15 E to Hold and Erase line 15.
n DELETE	Delete previous n characters before cursor.
n E	Erase line n by filling it with spaces.
F string	Find string starting at cursor until end of screen. If string is not found on screen, the cursor is placed at top of screen with a given error message.
FLUSH	Flush or write updated screen to disk. (part of OmniFORTH *)
n H	Hold line n by copying to PAD.

2.2 EDITOR Commands (continued)

COMMAND	DESCRIPTION
n I	Insert text from PAD at line n by pushing lines n+1 down and discarding last line.
n m INDEX	Index listing of line 0 of each screen beginning with screen n until screen m. (part of OmniFORTH *)
L	Re-List current edit screen. Use n LIST to list any other screen.
n LIST	List screen n and set SCR making n current edit screen. (part of OmniFORTH *)
n M	Move cursor by signed n characters and display PAD. If n<0 then move backwards.
N	Next occurence of previous Find string. If string is not found, the cursor is moved to top of screen and an error message given. This allows another scan of current screen.
n P string	Put string on line n and in PAD. This command is used to input and overlay lines of text within a screen.
n R	Replace on line n text held in PAD.
n S	Spread by moving line n downwards and space filling line n. Line 15 is lost.
n m SHOW	List screens in TRIADs (three per page). Uses TRIAD and includes screens n thru m in show list. (part of OmniFORTH *)
n T	Type line n, placing line in PAD and n on stack.
TILL string	Delete from cursor unTILL end of string.
TOP	Top cursor home to TOP of screen.
n TRIAD	List screens in TRIADS (three per page). Each page begins with a screen number evenly divisible by 2. TRIAD allows replacing one page, rather than listing entire application when changes are made.(part of OmniFORTH *)
X string	Delete the first occurence of string, starting scan from current cursor position until end of screen.

2.3 Entering the EDITOR

CAUTION: Make a backup copy of your disk using DOS before starting an edit session.

The source for the EDITOR is provided beginning on screen 36 of the OmniFORTH disk.

To determine if EDITOR is resident, type:

```
EDITOR
```

If the EDITOR is not resident you will see EDITOR? and will need to type in: 36 LOAD EDITOR followed by a carriage return to load the EDITOR and enter its vocabulary.

A prompt OK will appear when the EDITOR is entered and you will be in the EDITOR's vocabulary until you return to OmniFORTH by compiling a new word or requesting another vocabulary.

2.4 Screen Commands

The EDITOR works with a page of text called a screen. A screen consists of 16 lines (numbered 0-15) with 64 characters on each line. Screen numbers are assigned to each screen and represent a contiguous area on disk. OmniFORTH variable SCR is used to store the current editing screen number and will be altered by most of the screen commands. You may alter it yourself by typing:

```
n SCR !
```

where n is the screen number that you would like. Most of the screen commands will do this for you automatically.

2.4.1 Screen List: n LIST (part of OmniFORTH *)

Once you enter the EDITOR you should list a screen. Sample edit screens 57, 58, and 59 have been provided on the OmniFORTH disk to give you a chance to learn how to use the EDITOR. You may list a screen at any time by typing its number and LIST. For example,

```
57 LIST        or  58 LIST     or 59 LIST
```

will list screen 57,58, or 59. LIST not only lists a screen, it also places the screen number in the OmniFORTH variable SCR. The EDITOR uses SCR to determine which screen is being edited. You should LIST a screen before you begin editing it to verify that it is the one you want and to set SCR.

2.4.2 Screen Re-List: L

Once you are in the EDITOR's vocabulary, you can type L to list the current edit screen. Try typing:

```
L
```

to list the current edit screen stored at SCR.

2.4.3 Screen Index: n m INDEX (part of OmniFORTH *)

INDEX will list line 0 of a range of screens starting at screen n until screen m. INDEX is part of OmniFORTH and can be called at any time. For example type:

```
36 44 INDEX
```

to list the first line (line 0) of screens 36 thru 44. Note that comments are enclosed by left and right parentheses ( ) preceded and followed by a space, and can be used for documentation anywhere on screen. It is a good practice to use line 0 as a comment line to describe the screen. INDEX will not alter SCR.

2.4.4 Screen Triad List: n TRIAD (part of OmniFORTH *)

To list the entire text of a triad of screens, type:

```
36 TRIAD        or  37 TRIAD    or  38 TRIAD
```

which will list three screens per page, the first screen always starting with a number evenly divisible by three. This is done so you won't have to list an entire source file just to update a few page changes. TRIAD alters SCR to last screen listed on page.

2.4.5 Screen Show: n m SHOW (part of OmniFORTH *)

SHOW uses TRIAD to list entire screens over a range. For example type:

```
36 44 SHOW
```

to list triads of screens that make up the text EDITOR. Note that the first screen on each page is evenly divisible by three including screen n on first page and m on last page. SHOW alters SCR to last screen listed on page.

2.4.6 Screen Copy: n m COPY

COPY allows entire screens of text to be transfered from one screen to another. Text from screen n will be copied to screen m. For example type:

```
30 59 COPY
```

to copy screen 30 to 59. Note that SCR will not be altered by COPY.

2.4.7 Screen Clear: n CLEAR

CLEAR screen n to spaces and set SCR equal to n. Type:

```
59 CLEAR
```

to clear screen 59 and set SCR to 59 making it the current edit screen.

2.4.8 Screen Flush: FLUSH (part of OmniFORTH *)

FLUSH is a part of OmniFORTH and can be called any time to write an updated screen to disk. FLUSH requires no parameters and is called by:

```
FLUSH
```

Note that FLUSH will only write to disk if there has been an update to a screen. FLUSH will not alter SCR.

2.5 Line Commands

The EDITOR contains a vocabulary of line editing commands that allows complete control text lines. There are 64 characters per line and 16 lines (numbered 0 thru 15) in an edit screen.

Please copy screen 57 to 59 and use 59 as a sample edit screen to familiarize yourself with the line commands. Type:

```
57 59 COPY   ( to copy screen 57 to 59 )

TOP          ( to position cursor to TOP )

59 LIST      ( to list screen 59 and set SCR=59 )
```

2.5.1 Line Delete: n D

Delete line n of current edit screen by moving line to PAD and pulling lines n+1 thru 15 up, reproducing line 15. Example:

```
5 D
```

will hold line 5 in PAD, pull lines 6 thru 15 up into 5 thru 14 thus reproducing line 15 at 14. Remember that D will renumber the remaining lines in the screen each time it is used. Observe that 15 D will not work because 15+1 is off current edit screen. Use 15 H then 15 E to hold and erase line 15. Note that PAD holds original line 5 ready for other line commands like I or R.

2.5.2 Line Erase: n E

Erase line n by filling it with spaces. Example:

```
12 E
```

will space or blank fill entire 64 characters of line 12.

2.5.3 Line Hold: n H

Hold line n by copying it to PAD. For example:

```
15 H
```

will move line 15 into PAD without disturbing original line 15 in any way. PAD can be used by other line commands like I or R to reproduce PAD text on current or any other screen.

2.5.4 Line Insert: n I

Insert text held in PAD at line n by pushing lines n+1 down and discarding last line. Try:

```
8 I
```

to move line 8 thru 14 down to 9 thru 15 and then replace text held in PAD on line 8. Note that I will renumber the remaining lines in the screen each time it is used.

2.5.5 Line Put: n P string

Put string of text into PAD and copy to line n. Put is the command used to input new lines of text. For example:

```
12 P the quick brown fox
```

will move "the quick brown fox" into PAD and replace line 12 with the contents of PAD.

2.5.6 Line Replace: n R

Replace line n with the contents of PAD. For example, if PAD had been loaded using D, H, I, P, or T then typing:

```
7 R
```

will copy text from PAD onto line 7 of current edit screen. Note that R is useful to repeat lines on screens.

2.5.7 Line Spread: n S

Spread opens up line n by pushing lines n thru 14 down to n+1 thru 15 and then space filling line n. For example:

```
4 S
```

will push lines 4 thru 14 to lines 5 thru 15 and then blank fill line 4. Note that line 15 is lost and that each time S is used, the remaining lines in the screen are renumbered.

2.5.8 Line Type: n T

Type will display just one line of a screen placing the line of text in PAD and line number on stack. For example:

```
0 T
```

will type line 0, place its text in PAD, and line number 0 on the stack. Note that T can be used to set up other line editing commands D, E, I, R, and S. Remember that during a normal edit session you will probably use several T's, pushing numbers onto stack and not using or popping them off, so it is a good idea to clean up the stack occasionally by typing SP! followed by a carriage return.

2.6 String Commands

The EDITOR's vocabulary contains command definitions that allow interactive character strings and cursor manipulations. The cursor position is shown by the _ underline character. Note that one space must separate a command that uses a string parameter and all commands require a carriage return to execute.

Please copy screen 57 to 59 and use 59 as a sample edit screen to familiarize yourself with the line commands. Type:

```
57 59 COPY    ( to copy screen 57 to 59 )

TOP           ( to position cursor to TOP )

59 LIST       ( to list screen 59 and set SCR=59 )
```

2.6.1 String Backup: B

Backup cursor by the length of text in PAD. This command is normally used after a C, F, or N to reposition cursor to the beginning of the string. For example, after using F string, you may type:

```
B
```

to move cursor back to beginning of target string. Note that repeated use of B will backup cursor past TOP.

2.6.2 String Copy: C string

Copy string by inserting it into line at present cursor location. For example, after positioning cursor, you may type:

```
C copy string test
```

to insert "copy string test" at cursor location. The remainder of line is displaced right causing any trailing characters past the 64th to be lost. Remember to inspect the prompt line after using C to check if you have lost last few characters. If end of edit screen is detected, an error message OFF CURRENT EDITING SCREEN is given (use TOP to reset).

2.6.3 String Delete: n DELETE

To delete the previous n characters that appear before the cursor. For example if cursor is located after a string just type:

```
2 DELETE
```

and you will delete the last two characters of string and pull the remainder of line left. Note that if you delete while the cursor is positioned at beginning of a line you will pull line up into previous line.

2.6.4 String Find: F string

Find sting starting scan at cursor until end of screen. Try:

```
F target
```

to Find "target" string. If target string is not found or search encounters end of screen an error message is given and the cursor is placed at TOP ready for another scan. You may use N to search for Next target string.

2.6.5 String Move: n M

Move cursor by signed n locations and then display prompt line. For example:

```
-5 M    or  7 M
```

will move cursor back 5 or forward 7 positions and display line showing new cursor location. Note 0 M will display line without moving cursor.

2.6.6 String Next: N

Next will search for the Next occurence of the previous Find target string. Search starts at current cursor location and continues until end of screen. For example, after using F, try:

```
N
```

to find Next target used by F. Note if scan encounters end of screen an error message is given and the cursor is positioned to TOP ready for another scan.

2.6.7 String Till: TILL string

Deletes characters starting at cursor and continuing unTILL end of string has been deleted on current edit line. Position cursor and type:

```
TILL test
```

to delete all characters unTILL end of string. Note that an error message is given if srting is not found or end of line is encountered.

2.6.8 String Top: TOP

TOP will move cursor location home to the TOP left position of the current editing screen. TOP can be used to reset error conditions and to resume edit session.

2.6.9 String eXtract: X string

Remove or eXtract the first occurence of target string starting scan at current cursor location. Try:

```
X test
```

to eXtract the string "test" from the edit screen. Note that an error message is given if string is not found. The cursor is positioned to the TOP of the current edit screen to allow another scan.

OmniFORTH

Assembler

INTERACTIVE COMPUTER SYSTEMS, INC.
6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

3.1 OmniFORTH 8080 + Z80 Assembler

OmniFORTH provides the ability to implement native machine code of the resident processor. Although OmniFORTH provides power almost equal to assembly language there are some cases where the user may desire to use native machine code such as calling predefined subroutines or in speed critical routines. Assembly code is easy to produce in OmniFORTH, often easier than using a standard assembler.

OmniFORTH comes with a complete incremental structured assembler for each processor implemented. This implemenation is supplied with a 8080 assembler loaded and ready to use, plus and optional Z80 assembler that can be loaded by the user as needed.

3.2 CODE words

Users of FORTH refer to the assembly language routines as CODE words. CODE words once created may be used the same as any other OmniFORTH word. The OmniFORTH assembler is always resident and is invoked by the words CODE or ASSEMBLER. ASSEMBLER is the name of the assembly vocabulary just as FORTH is the name of the fundamental vocabulary.

CODE words begin with the word CODE and end with NEXT JMP. NEXT being the address of the reentry into OmniFORTH. The structure of a CODE word is:

CODE name . assembly code . NEXT JMP

where name is the name of word the same as used with a colon ":" definition.

Inorder to be complete, OmniFORTH provides the ability to loop and test by the use of the words BEGIN, UNTIL, IF ELSE and ENDIF (similar to their use in high level OmniFORTH).

There is one fundamental difference when coding code words from coding colon words, and that is that the interpreter is in execution mode and not in compile mode. Therefore high level OmniFORTH words such as SWAP, DROP, 1+, etc, may be used at assembly time to manipulate addresses, values etc. Another consideration is that like high level OmniFORTH, the reverse polish notation holds, therefore mnemonic definition is reversed from that of a standard assembler. An example of this can be seen in the NEXT JMP which would be JMP NEXT with a standard assembler.

3.3 Conditional Test Operators

When using the BEGIN ... UNTIL and IF ... ELSE ... ENDIF structures in the assembler, the UNTIL and IF statements will code conditional Jumps. Inorder to make the proper Jump code, the user must precede the IF or UNTIL by one of the assembler conditional test operators.

ASSEMBLER TEST OPERATORS

0=	True if condition code Z Bit is set
CS	True if condition code C Bit is set
PE	True if condition code P/V Bit is set
0<	True if condition code S Bit is set
NOT	Reverses logic of the above conditions

3.4 Code word terminations

Code words are terminated by Jumping into the interpreter, therefore the OmniFORTH assembler provides three constants that provide absolute addresses within the interpreter they are:

NEXT	Standard entry into the interpreter
HPUSH	Push's the HL register on the stack then goes to NEXT
WHPUSH	Push's the WW' (or DE) and the HL registers on the stack then goes to NEXT

These constants are used as:

```
   NEXT JMP
   HPUSH JMP
or WHPUSH JMP
```

3.5 OmniFORTH Register Designations

OmniFORTH uses two of the 16 bit register pairs to hold system pointers. They are the DE register which contains the word pointer and the BC register which contains the interpreter pointer, therefore most FORTH programmers prefer to refer to the D and E registers as W and W' and the B and C registers as the I and I'. For those die hard assembly programmers the user may load the D, E, B, and C operators from the optional instruction set.

The WW' register pair may be changed within a code word since NEXT will restore the WW' register pair.

WARNING:

The II' register pair must be restored by the user within the code word, if used, because FORTH will blow when NEXT is entered and if II' is incorrect.

The user may use all other registers (except the stack pointer SP) since all FORTH intra word parameters are passed on the stack.

3.6 Optional Z80 Instruction Set

Inorder to save memory space the entire Z80 instruction set is not precompiled in OmniFORTH the user is provided with the source of the remaining mnemonics and may elect to load any part or all. The optional mnemonics are indicated in the mnemonic list by an asterisk in column one.

3.7 Compatibility

Inorder to remain compatible with other versions of FORTH in 8 bit immediate instructions, OmmniFORTH allows the use of the immediate operator #. The following table lists alternate forms of some mnemonics listed in the assembly mnemonic table:

Standard	Alternate (available in OmniFORTH)
yy R MVI	yy # R MOV
yy ADI	yy # ADD
yy ACI	yy # ADC
yy ANI	yy # ANA
yy XRI	yy # XRA
yy CPI	yy # CMP
yy ORI	yy # ORA
yy SUI	yy # SUB
yy SBI	yy # SBB

Normally CODE words are written in HEX mode therefore the HEX values A,B,C,D,& E would be misinterpeted as registers instead of the HEX values. Inorder to avoid this problem, the user when using the assembler should use 0A, 0B, 0C, 0D, & 0E for the HEX values.

WARNING:

The B, C, D, & E register mnemonics are not resident in the standard system therefore if use is desired they must be loaded from the optional assembly mnemonic source.

This manual is not intended to teach assembly language programming, but to be a guide in implementing machine code in FORTH. User's that are not already proficient in assembly language coding should consult other reference material.

3.8 Examples

The following are examples of simple code words used to illustrate OmniFORTH assembly coding. It should be noted that they are for example purposes only and may already exist or may be of no practical use.

3.8.1 Duplicate the top stack item.

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

3.8.2 Add the top 2 stack items.

```
CODE +   W POP   H POP   W DAD   HPUSH JMP
```

3.8.3 SWAP the top 2 stack items.

```
CODE SWAP   H POP   XTHL   HPUSH JMP
```

3.8.4 Call a user routine located at address F000.

```
CODE SAM   F000 CALL   NEXT JMP
```

3.8.5 Search memory starting at the address on top of the stack for a character contained in the second item on the stack, leaving the address of the character on the stack

```
CODE SEARCH-MEM   H POP   W POP   W' A MOV
                  BEGIN   M CMP   H INX   0= UNTIL
                  H DCX   H PUSH   NEXT JMP
```

3.8.6 Replace the top of stack with a 1 (TRUE) if it is equal to a zero otherwise replace it with a 0 (FALSE) .

```
CODE SAM   H POP   A XRA   H ORA   L ORA   0 H MVI
           0= IF   1 L MVI   ELSE
           0 L MVI   ENDIF   HPUSH JMP
```

3.9 Assembler Mnemonics

The following mnemonic table was modified and reprinted from A.M. Ashley's PDS Assembly Language Development System which was used to develop this Z80/8080 OmniFORTH System.

Reprinted with thanks and permission of:

A.M. Ashley
395 Sierra Madre Villa
Pasadena, CA 91107

(213) 793-5748

3.9.1 Register Mnemonics

All of the Z80 registers have been assigned predefined mnemonics. These assignments agree with those given by INTEL and ZILOG.

The predefined register set is defined as:

	Register	Definition	Value
	A	Accumulator	7
*	B	8 or 16 bit	0
	I	8 or 16 bit	0
*	C	8 bit	1
	I'	8 bit	1
*	D	8 or 16 bit	2
	W	8 or 16 bit	2
*	E	8 bit	3
	W'	8 bit	3
	H	8 or 16 bit	4
	L	8 bit	5
	M	Memory Indirect (HL)	6
	SP	Stack Pointer	6
	PSW	Program Status Word	6
	IX	16 bit Index	0DDH
	IX)	16 bit Index	0DDH
	IY	16 bit Index	0FDH
	IY)	16 bit Index	0FDH
*	RF	Refresh Register	04FH
*	IV	Interrupt Vector	047H
	#	Immediate	

These register assignments may not be redefined.

* Indicates an optional mnemonic (see text).

3.9.2 Assembly Language

As a consequence of favoring the INTEL mnemonic set over that of ZILOG, the Z80 instruction superset has been invented. One consideration in the definition of instruction mnemonics is standard assembly language convention. In the instruction mnemonics which follow.

nn pp	refers to an arbitrary 16 bit datum;
yy	refers to an arbitrary 8 bit datum;
d	refers to a Z80 displacement except for relative jumps;
R	refers to an 8 bit register (A, B or I, C or I', D or W, E or W', H, L, M)
RP	refers to an 16 bit register pair (B or I, D or W, H, SP)
QP	refers to an 16 bit register pair (PSW, B or I, D or W, H)

3.9.3 Eight Bit Load

MNEMONIC	ZILOG	REMARKS
R R MOV	LD R,R	From register to register
d IX R MOV	LD R,(IX+d)	Register indirect (R not= M)
d IY R MOV	LD R,(IY+d)	
R d IX MOV	LD (IX+d),R	Memory indirect (R not= M)
R d IY MOV	LD (IY+d),R	
IV A MOV	LD A,I	Fetch interrupt vector
RF A MOV	LD A,R	Fetch refresh register
A IV MOV	LD I,A	Load interrupt vector
A RF MOV	LD R,A	Load refresh register

3.9.4 Accumulator Load / Store

nn pp LDA	LD A,(nn)	Accumulator direct
B LDAX	LD A,(BC)	Accumulator extended
D LDAX	LD A,(DE)	
nn pp STA	LD (nn),A	Accumulator direct
B STAX	LD (BC),A	Accumulator extended
D STAX	LD (DE),A	

3.9.5 Eight Bit Load Immediate

MNEMONIC	ZILOG	REMARKS
yy # R MOV	LD R,n	Register immediate
* yy R MVI	LD R,n	Register immediate
yy d IX MVI	LD (IX+d),n	Memory indirect immediate
yy d IY MVI	LD (IY+d),n	

3.9.6 Sixteen Bit Load / Store

qq pp RP LXI	LD RP,nn	Extended immediate
qq pp IX LXI	LD IX,nn	
qq pp IY LXI	LD IY,nn	
qq pp LHLD	LD HL,(nn)	Extended indirect load
* qq pp LBCD	LD BC,(nn)	
* qq pp LDED	LD DE,(nn)	
* qq pp LIXD	LD IX,(nn)	
* qq pp LIYD	LD IY,(nn)	
* qq pp LSPD	LD SP,(nn)	
qq pp SHLD	LD (nn),HL	Extended indirect store
* qq pp SBCD	LD (nn),BC	
* qq pp SDED	LD (nn),DE	
* qq pp SIXD	LD (nn),IX	
* qq pp SIYD	LD (nn),IY	
* qq pp SSPD	LD (nn),SP	
SPHL	LD SP,HL	Set stack pointer
* SPIX	LD SP,IX	
* SPIY	LD SP,IY	
QP PUSH	PUSH QP	To stack
IX PUSH	PUSH IX	
IY PUSH	PUSH IY	
QP POP	POP QP	From stack
IX POP	POP IX	
IY POP	POP IY	

* Indicates an optional mnemonic (see text).

3.9.7 Exchange, Block Transfers, and Search

	MNEMONIC	ZILOG	REMARKS
	XCHG	EX DE,HL	Exchange
*	EX	EX AF,AF'	
*	EXX	EXX	
	XTHL	EX (SP),HL	
*	XTIX	EX (SP),IX	
*	XTIY	EX (SP),IY	
*	LDI	LDI	Transfer
	LDIR	LDIR	
*	LDD	LDD	
	LDDR	LDDR	
*	CPD	CPD	Search
*	CPDR	CPDR	
*	CPII	CPI	
	CPIR	CPIR	

3.9.8 Eight Bit Arithmetic and Logical

	R ADD	ADD R	Add register
	yy # AND	ADD A,yy	Add immediate
*	yy ADI	ADD A,yy	Add immediate
	d IX ADD	ADD (IX+d)	Add indirect
	d IY ADD	ADD (IY+d)	
	R ADC	ADD R	Register with carry
	d IX ADC	ADC (IX+d)	Memory indirect with carry
	d IY ADC	ADC (IY+d)	
	yy # ADD	ADC n	Immediate with carry
*	yy ACI	ADC n	Immediate with carry

* Indicates an optional mnemonic (see text).

3.9.8 Eight Bit Arithmetic and Logical (Continued)

MNEMONIC	ZILOG	REMARKS
R SUB	SUB R	Subtract register
d IX SUB	SUB (IX+d)	Subtract memory indirect
d IY SUB	SUB (IX+d)	
R SBB	SBC R	Register with carry
d IX SBB	SBC (IX+d)	Memory indirect with carry
d IY SBB	SBC (IY+d)	
R ANA	AND R	Logical and register
d IX ANA	AND (IX+d)	Memory indirect
d IY ANA	AND (IY+d)	
R ORA	OR R	Logical OR register
d IX ORA	OR (IX+d)	Memory indirect
d IY ORA	OR (IY+d)	
R XRA	XOR R	Exclusive OR register
d IX XRA	XOR (IX+d)	Memory indirect
d IY XRA	XOR (IY+d)	
R CMP	CP R	Register compare
d IX CMP	CP (IX+d)	Memory indirect
d IY CMP	CP (IY+d)	
R INR	INC R	Register increment
d IX INR	INC (IX+d)	
d IY INR	INC (IY+d)	
R DCR	DEC R	Register decrement
d IY DCR	DEC (IX+d)	
d IY DCR	DEC (IY+d)	
yy # ANA	AND yy	Accumulator immediate
* yy ANI	AND yy	
yy # XRA	XOR yy	
* yy XRI	XOR yy	
yy # CMP	CP yy	
* yy CPI	CP yy	
yy # ORA	OR yy	
* yy ORI	OR yy	
yy # SUB	SUB yy	
yy SUI	SUB yy	
yy # SBB	SBC A,yy	
* yy SBI	SBC A,yy	

* Indicates an optional mnemonic (see text).

3.9.9 General Purpose Arithmetic and CPU Control

MNEMONIC	ZILOG	REMARKS
DAA	DAA	Decimal adjust accumulator
CMA	CPL	Complement accumulator logical
NEG	NEG	Negate accumulator
CMC	CCF	Compliment carry flag
STC	SCF	Set carry flag
NOP	NOP	No operation
HLT	HALT	Halt CPU
DI	DI	Disable interrupts
EI	EI	Eable interrupts
0 IM	IM 0	Set interrupt mode
1 IM	IM 1	
2 IM	IM 2	

3.9.10 Sixteen Bit Arithmetic Group

RP DAD	ADD HL,RP	16 bit add (RP not= H, IY)
RP CAD	ADC HL,RP	Add with carry (RP not= H, IY)
RP SBC	ADD IX,RP	Add register pair to IX
RP IY DAD	ADD IY,RP	Add register pair to IY
RP INX	INC RP	16 bit increment
IX INX	INC IX	
IY INX	INC IY	
RP DCX	DEC RP	16 bit decrement
IX DCX	DEC IX	
IY DCX	DEC IY	

3.9.11 Rotate and Shift Group

RLC	RLCA	Accumulator left circular
RAL	RLA	Left circular through carry
RRC	RRCA	Accumulator right circular
RAR	RRA	Right circular through carry
R SCL	RLC R	Register left circular
* M SCL	RLC (HL)	Memory left circular
* d IX SCL	RLC (IX,d)	Left circular memory indirect
* d IY SCL	RLC (IY,d)	
* R RL	RL R	Register left through carry
* R SRC	RRC R	Register right circular
* R RR	RR R	Register right through carry
* R SLA	SLA R	Left linear bit 0 = 0
* R SRA	SRA R	Right linear bit 7 = extended
* R SRL	SRL R	Right linear bit 7 = 0
* RLD	RLD	Left decimal
* RRD	RRD	Right decimal

* Indicates an optional mnemonic (see text).

3.9.12 Bit Manipulation (b=bit number 0 <= b <= 7)

	MNEMONIC	ZILOG	REMARKS
*	R b BIT	BIT b,R	Zero flag = bit b of R
*	M b BIT	BIT b,(HL)	
*	d IX b BIT	BIT b,(IX+d)	
*	d IY b BIT	BIT b,(IY+d)	
*	R b STB	SET b,R	SET (1) bit b of R or memory
*	M b STB	SET b,(HL)	
*	d IX b STB	SET b,(IX+d)	
*	d IY b STB	SET b,(IY+d)	
*	R b RES	RES b,R	Reset (0) bit b of R or memory
*	M b RES	RES b,(HL)	
*	d IX b RES	RES b,(IX+d)	
*	d IY b RES	RES b,(IY+d)	

3.9.13 Input / Output Group (P=Port number R=Register)

	P IN	IN A,(P)	Input to accumulator
	R CIN	IN R,(C)	Register R from port (C)
*	INI	INI	Input and increment
*	INIR	INIR	Repeat input and increment
*	IND	IND	Input and decrement
*	INDR	INDR	Repeated input and decrement
	P OUT	OUT (P),A	Output accumulator
	R COUT	OUT (C),R	Register R to port (C)
*	OUTI	OUTI	Output and increment
*	OUTIR	OUTIR	Repeated output and increment
*	OUTD	OUTD	Output and decrement
*	OUTDR	OUTDR	Repeated output and decrement

* Indicates an optional mnemonic (see text).

3.9.14 Jump Group (V=address dest=destination +-128 bytes)

MNEMONIC	ZILOG	REMARKS
V JMP	JP V	Jump
* V JNC	JP NC,V	No carry
* V JC	JP C,V	Carry
* V JNZ	JP NZ,V	Not zero
* V JZ	JP Z,V	Zero
* V JPO	JP PO,V	Parity odd
* V JPE	JP PE,V	Parity even
* V JP	JP P,V	Positive
* V JM	JP M,V	Negative
* dest JR	JP d	Jump relative
* dest JRC	JR C,d	Carry
* dest JRNC	JR NC,d	No carry
* dest JRZ	JR Z,d	Zero
* dest JRNZ	JR NZ,d	Not zero
PCHL	JP (HL)	Branch to location in HL
* PCIX	JP (IX)	Branch to location in IX
* PCIY	JP (IY)	Branch to location in IY
* dest DJNZ	DJNZ,d	Decrement and jump if not zero

3.9.15 Call and Return Group (V=address)

V CALL	CALL V	Subroutine transfer
* V CNC	CALL NC,V	No carry
* V CC	CALL C,V	Carry
* V CNZ	CALL NZ,V	Not zero
* V CZ	CALL Z,V	Zero
* V CPE	CALL PE,V	Parity even
* V CPO	CALL PO,V	Parity odd
* V CP	CALL P,V	Positive
* V CM	CALL M,V	Negative
* RET	RET	Return
* RNC	RET NC	No carry
* RC	RET C	Carry
* RNZ	RET NZ	Not Zero
* RZ	RET Z	Zero
* RPE	RET PE	Parity even
* RPO	RET PO	Parity odd
* RP	RET P	Positive
* RM	RET M	Negative
RETI	RETI	Return from interrupt
RETN	RETN	Return from non-maskable interrupt
n RST	RST n	Restart

* Indicates an optional mnemonic (see text).

OmniFORTH

Glossary

INTERACTIVE COMPUTER SYSTEMS, INC.
6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

This glossary contains most of the word definitions that have been released with OmniFORTH. Please note that a particular implementation of OmniFORTH may not include all of the words shown in this glossary. Words that are illustrated are from fig-FORTH Release 1 plus additional words have been added to provide the user with a more powerful vocabulary.

The definitions are presented in the order of their ASCII sort.

Unless otherwise noted, all references to numbers are for 16 bit signed integers in a stack that is 16 bits wide. Double integers are 32 bits long and take two stack locations, the most significant part including sign is on top of the stack.

The glossary illustrates the first line of each entry with the word followed by a description of the action of the procedure on the stack. The symbols indicate the order in which input parameters have been placed on the stack. In this notation, the top of the stack is to the right. Three dashes "---" indicate the execution point and any parameters left on the stack are listed.

Symbols include:

addr	16 bit memory address
b	8 bit byte with zeros in upper 8 bits
c	7 bit ASCII character with zeros in upper 9 bits
d	32 bit signed double integer with most significant part including sign on top of stack
f	Boolean flag. 0=false, non-zero=true
ff	Boolean false flag. f=0
n	16 bit signed integer number
u	16 bit unsigned integer number
tf	Boolean true flag. f=non-zero

The capital letters shown on the right indicate FORTH definition characteristics:

C	May only be used within a colon definition. A digit indicates number of memory addresses used.
E	Intended for execution only.
L0	Level zero definition of FORTH-78.
L1	Level one definition of FORTH-78.
P	Has precedence bit set. Will execute even when compiling.
U	A user variable.

✓ ! n addr --- LO

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

✓ !CSP ---

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

✓ # d1 --- d2 LO

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

✓ #> d --- addr count LO

Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#BUF --- n

A constant returning the number of disk buffers allocated. For the disk I-O routines to work correctly, #BUF must be greater than 1.

✓ #D ---

Special output formatting. Converts one decimal digit and holds it in PAD.

✓ #H

Special output formatting. Converts one HEX digit and holds it in PAD.

✓ #S d1 --- d2 LO

Generates ascii text in the text output buffer, by the use of #, until a zero double number n2 results. Used Used between < # and # > .

✓ ' --- addr P,LO

Used in the form:

' nnnn

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

✓ (--- P,LO

Used in the form:

(cccc)

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

✓ (.") --- C+

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

✓ (;CODE) --- C

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

✓ (+LOOP) n --- C2

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

✓ (ABORT) ---

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

✓ (DO) --- C

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

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

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

✓ (LINE) n1 n2 --- addr count

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

✓ (LOOP) --- C2

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

✓ (NUMBER) d1 addr1 --- d2 addr2

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

✓ * n1 n2 --- prod LO

Leave the signed product of two signed numbers.

✓ */ n1 n2 n3 --- n4 LO

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

n1 n2 * n3 /

✓ */MOD n1 n2 n3 --- n4 n5 LO

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

✓ + n1 n2 --- sum LO

Leave the sum of n1+n2.

✓ +! n addr --- LO

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

✓ +- n1 n2 --- n3

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

✓ +BUF add1 --- addr2 f

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

✓ +LOOP n1 --- (run)
addr n2 --- (compile) P,C2,LO

Used in a colon-definition in the form:

DO ... n1 +LOOP

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

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

✓ +ORIGIN n --- addr

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

✓ , n --- LO

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

✓ - n1 n2 --- diff LO

Leave the difference of n1-n2.

✓ --> --- P,LO

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

✓ -DUP

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

LO

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

✓ -FIND

--- pfa b tf (found)
--- ff (not found)

Accepts the next text word (delimited by blanks) in the input strean to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, is length byte, and a boolean true is left. Otherwise, only a boolean false is left.

✓ -TRAILING

addr n1 --- addr n2

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

✓ .

n ---

LO

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

✓ ..

n ---

Outputs n as an unsigned number per the present BASE.

✓ ."

P,LO

Used in the form:

." cccc"

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

✓ .2H n ---

Outputs to console n as two HEX digits. Any overflow is lost.

✓ .4H n ---

Outputs to console n as four HEX digits.

✓ .CPU

Prints the processor name (i.e., 8080) from ORIG+22H encoded as a 32 bit, base 36 integer.

✓ .LINE line scr ---

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

✓ .R n1 n2 ---

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

✓ / n1 n2 --- quot LO

Leave the signed quotient of n1/n2.

✓ /MOD n1 n2 --- rem quot LO

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

✓ 0 1 2 3 --- n

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

✓ 0< n --- f L0

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

✓ 0= n - f L0

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

✓ OBRANCH f --- C2

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

✓ 1+ n1 --- n2 L1

Increment n1 by 1.

✓ 2+ n1 --- n2

Leave n1 incremented by 2.

✓ 2! nlow nhigh addr ---

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

✓ 2@ addr --- nlow nhigh

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

✓ 2DUP n2 n1 --- n2 n1 n2 n1

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

: --- P,E,LO

Used in the form called a colon-definition:

```
: cccc   ...   ;
```

Creates a dictionary entry defining cccc as equivalent to the following sequence of FORTH word definitions '...' until the next ';' or ';CODE'. The compiling process is done by the text interpreter as long as STATE is non-zero. Other details are that the CONTEXT vocabulary is set to the current vocabulary and that words with the precedence bit set (P) are executed rather than being compiled.

; --- PcC,LO

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

;CODE --- P,C,LO

Used in the Form:

```
: cccc   ....   ;CODE assembly mneumonics
```

Stop compilation and terminate a new defining word cccc by compiling (;CODE). Set the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following mneumonics.

When cccc later executes in the form:

```
cccc   nnnn
```

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

;S --- P,LO

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

✓ < n1 n2 --- f LO

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

✓ <# --- LO

Setup for pictured numeric output formatting using the words:

```
< # # #S SIGN #>
```

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

✓ <BUILDS --- C,LO

Used within a colon-definition:

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

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

```
cccc nnnn
```

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

✓ = n1 n2 --- f LO

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

✓ > n1 n2 --- f LO

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

✓ >R n --- C,LO

Remove a number from the computation stack and place as the most accessible on the return stack. Use should be balanced with R> in the same definition.

✓ ? addr -- LO

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

✓ ?COMP ---

Issue error message if not compiling.

✓ ?CSP ---

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

✓ ?ERROR f n ---

Issue an error message number n, if the bolean flag is true.

✓ ?EXEC ---

Issue an error message if not executing.

✓ ?LOADING ---

Issue an error message if not loading.

✓ ?PAIRS n1 n2 ---

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

✓ ?S ---

Outputs the contents of the parameter stack in HEX and DECIMAL.

✓ ?STACK ---

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

✓ ?TERMINAL --- f

Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation. This definition is installation dependent.

✓ @ addr --- n LO

Leave the 16 bit contents of address.

✓ ABORT --- LO

Clear the stacks and enter the execution state. Return control of the operators terminal, printing a message appropriate to the installation.

✓ ABS n - u LO

Leave the absolute value of n as u.

✓ AGAIN addr n --- (compiling) P,C2,LO

Used in a colon-definition in the form:

```
BEGIN ... AGAIN
```

At run-time, AGAIN forces execution to return to corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).

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

✓ ALLOT n --- LO

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

✓ AND n1 n2 --- n2 LO

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

✓ B/BUF --- n

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

✓ B/SCR --- n

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

✓ BACK addr ---

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

✓ BASE --- addr U,LO

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

✓ BEGIN --- addr n (compiling) P,LO

Occurs in a colon-definition in form:

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

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

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

✓ BL --- c

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

✓ BLANKS addr count ---

Fill an area of memory beginning at addr with blanks.

✓ BLK --- addr U,LO

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

✓ BLOCK n --- addr LO

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

✓ BRANCH --- C2,LO

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

✓ BUFFER n --- addr

Obtain the next memory buffer, assigning it to block n. If the contents of the buffer is marked as updated, it is written to the disc. The block is not read from the disc. The address left is the first cell within the buffer for data storage.

✓ C! b addr ---

Store 8 bits at address. On word addressing computers, further specification is necessary regarding byte addressing.

✓ C, b ---

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

✓ C/L --- n

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

✓ C@ addr --- b

Leave the 8 bit contents of memory address. On word addressing computers, further specification is needed regarding byte addressing.

✓ CFA pfa --- cfa

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

✓ CMOVE from to count ---

Move specified quantity of bytes beginning at address (from) to address (to). The contents of address from is moved first proceeding toward high memory. Further specification is necessary on word processing computers.

✓ COLD ---

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

✓ COMPILE ---

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

✓ CONSTANT n --- LO

A defining word used in the form:

n CONSTANT cccc

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

✓ CONTEXT --- addr U,LO

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

✓ COUNT addr1 --- addr2 n LO

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

✓ CR --- L0

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

✓ CREATE ---

A defining word used in the form:

CREATE cccc

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

✓ CSP --- addr U

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

✓ D+ d1 d2 --- dsum

Leave the double number sum of two double numbers

✓ D+- d1 n --- d2

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

✓ D. d --- L1

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

✓ D.R d n ---

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

✓ DABS d --- ud

Leave the absolute value ud of a double number.

✓ DECIMAL --- LO

Set the numeric conversion BASE for decimal input-output.

✓ DEFINITIONS --- L1

Used in the form:

cccc DEFINITIONS

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

✓ DENSITY --- addr

A variable used by the disk interface.
0 = single density; 1 - double density.

✓ DIGIT c n1 --- n2 tf (ok)
c n1 --- ff (bad)

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

✓ DISK-ERROR --- addr

A variable used by the disk interface, containing the disk status for the last sector read or written.
0 means no error.

DLIST ---

List the names of the dictionary entries in the CONTEXT vocabulary.

✓ DLITERAL

d --- d (executing)
d --- (compiling) P

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

✓ DMINUS

d1 --- d2

Convert d1 to its double number two's complement.

✓ DO

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

Occurs in a colon-definition in form:

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

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

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

✓ DOES>

--- LO

A word which defines the run-time action within a high-level defining word. DOES > alters the code of field and first parameter of the new word to execute the sequence of compiled word addresses following DOES> . Used in combination with <BUILDS. When the DOES> part executes it begins with the address of the first parameter of the new word on the stack. This allows interpretation using this area or its contents. Typical uses include the FORTH assembler, multi-dimensional arrays, and compiler generation.

✓ DP ---- addr U,L

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

✓ DPL ---- addr U,LO

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

✓ DR0 ---
✓ DR1 ---

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

✓ DRIVE --- addr

A variable used by disk interface, containing the disk drive number (0 to MXDRV) used on the last sector read or written.

✓ DROP n --- LO

Drop the number from the stack.

✓ DUMP addr n --- LO

Print the contents of n memory locations beginning at addr. Both addresses and contents are shown in HEX and ASCII.

✓ DUP n --- n n LO

Duplicate the value on the stack.

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

Occurs within a colon-definition in the form:

IF ... ELSE ... ENDIF

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

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

EMIT c --- LO

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

EMPTY-BUFFERS --- LO

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

ENCLOSE addr1 c ---

ddr1 n1 n2 n3

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

END --- P,C2,LO

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

✓ ENDIF addr n - (compile) P,CO,LO

Occurs in a colon-definition in form:

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

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

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

✓ ERASE addr n ---

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

✓ ERROR line --- in blk

Execute error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to screen 27 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 27. If WARNING=0, n is just printed as a message number (non disc installation). If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT) . fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

✓ EXECUTE addr ---

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

✓ EXPECT addr count --- LO

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

✓ FENCE --- addr U

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

✓ FILL addr quan b ---

Fill memory at the address with the specified quantity of bytes b.

✓ FIRST --- n

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

✓ FLD --- addr U

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

✓ FLUSH

Write all UPDATEd disk buffers to disk. Should be used after editing, before dismounting a disk, or before exiting FORTH.

✓ FORGET --- E,LO

Executed in the form:

FORGET cccc

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

✓ FORTH --- P,L1

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

✓ HERE --- addr LO

Leave the address of the next available dictionary location.

✓ HEX --- LO

Set the numeric conversion base to sixteen (hexadecimal).

✓ HLD --- addr LO

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

✓ HOLD c --- LO

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

✓ I --- n C,LO

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

✓ ID. addr ---

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

✓ IF f --- (run-time)

--- n (compile) P,C2,LO

Occurs in a colon-definition in the forms:

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

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

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

✓ IMMEDIATE ---

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

✓ IN --- addr LO

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

✓ INDEX from to ---

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

✓ INTERPRET ---

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

✓ KEY --- c LO

Leave the ascii value of the next terminal key struck.

✓ LATEST --- addr

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

✓ LEAVE --- C,LO

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

✓ LFA pfa --- lfa

Convert the parameter field address of a dictionary definition to its link field address.

✓ LIMIT ---- n

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

✓ LIST n --- LO

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

✓ LIT --- n C2,LO

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

✓ LITERAL n --- (compiling) P,C2,LO

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

```
: xxx calculate LITERAL ;
```

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

✓ LOAD n --- LO

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

✓ LOOP addr n --- (compiling) P,C2,LO

Occurs in a colon-definition in form:

```
DO ... LOOP
```

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

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

✓ M* n1 n2 --- d

A mixed magnitude math operation which leaves the double number signed product of two signed number.

✓ M/ d n1 --- n2 n3

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

✓ M/MOD ud1 u2 --- u3 ud4

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

✓ MAX n1 n2 --- max LO

Leave the greater of two numbers.

✓ MESSAGE n ---

Print on the selected output device the text of line n relative to screen 27 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number (disc un-available).

✓ MIN n1 n2 --- min LO

Leave the smaller of two numbers.

✓ MINUS n1 --- n2 LO

Leave the two's complement of a number.

✓ MOD n1 n2 --- mod LO

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

MODE ---

Outputs the contents of the user variable BASE in a BASE independent form.

MON ---

Exit to the system monitor, leaving a re-entry to FORTH, if possible.

MOVE addr1 addr2 n ---

Move the contents of n memory cells (16 bit contents) beginning at addr1 into n cells beginning at addr2. The contents of addr1 is moved first. This definition is appropriate on word addressing computers.

NEXT ---

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

NFA pfa --- nfa

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

NOOP

A Forth 'no operation'.

✓ NUMBER addr --- d

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

✓ OFFSET --- addr U

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

✓ OR n1 n2 -- or LO

Leave the bit-wise logical OR of two 16 bit values.

✓ OUT --- addr U

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

✓ OVER n1 n2 --- n1 n2 n1 LO

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

✓ P! b port# ---

8080 or Z-80 I-O port store. Outputs byte b to port#.

✓ P@ port# --- b

8080 or Z-80 I-O port fetch. Inputs byte b from port#.

✓ PAD --- addr LO

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

✓ PFA nfa --- pfa

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

POP ---

The code sequence to remove a stack value and return to NEXT. POP is not directly executable, but is a FORTH re-entry point after machine code.

✓ PREV ---- addr

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

PUSH ---

This code sequence pushes machine registers to the computation stack and returns to NEXT. It is not directly executable, but is a FORTH re-entry point after machine code.

PUT ---

This code sequence stores machine register contents over the topmost computation stack value and returns to NEXT. It is not directly executable, but is a FORTH re-entry point after machine code.

✓ QUERY ---

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

✓ QUIT --- L

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

✓ R --- n

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

✓ R# --- addr

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

✓ R/W addr blk f ---

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

✓ R> --- n LO

Remove the top value from the return stack and leave it on the computation stack. See >R and R.

✓ RO --- addr U

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

✓ REPEAT addr n --- (compiling) P,C2

Used within a colon-definition in the form:

BEGIN ... WHILE ... REPEAT

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

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

✓ ROT n1 n2 n3 --- n2 n3 n1 LO

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

✓ RP! ---

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

✓ RP@ --- addr

Leaves the current value in the return stack pointer register.

✓ S->D n --- d

Sign extend a single number to form a double number.

✓ S0 --- addr U

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

✓ SCR --- addr U

A user variable containing the screen number most recently referenced by LIST.

✓ SEC --- addr

A variable used by the disk interface, containing the sector number last read or written relative to the last drive used.

✓ SHOW n1 n2 ---

Outputs screens n1 through n2 with 3 screens per page. *used TRIAD)

✓ SIGN n d --- d LO

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

✓ SMUDGE ---

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

✓ SP! ---

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

✓ SP@ --- addr

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

✓ SPACE --- LO

Transmit an ASCII blank to the output device.

✓ SPACES n --- LO

Transmit n ascii blanks to the output device.

✓ STATE --- addr LO,U

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

✓ SWAP n1 n2 --- n2 n1 LO

Exchange the top two values on the stack.

✓ T&SCALC n ---

Track & Sector and drive calculation for disk IO. n is the total sector displacement from the first logical drive to the desired sector.

n = (block# + OFFSET) * SEC/BLK

The corresponding drive, track, and sector numbers are calculated. If the drive number is different from the contents of DRIVE, the new drive number is stored in DRIVE and SET-DRIVE is executed.

The track number is stored in TRACK; the sector number is stored in SEC. T&SCALC is executed by RWDSK.

✓ TASK ---

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

✓ THEN --- P,CO,LO

An alias for ENDIF.

✓ TIB --- addr U

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

✓ TOGGLE addr b ---

Complement the contents of addr by the bit pattern b.

✓ TRAVERSE addr1 n --- addr2

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

✓ TRIAD scr ---

Display on the selected output device the three screens which include that numbered scr, beginning with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen 27.

✓ TYPE addr count --- LO

Transmit count characters from addr to the selected output device.

✓ U* u1 u2 --- ud

Leave the unsigned double number product of two unsigned numbers.

✓ U/ ud u1 --- u2 u3

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

U u1 u2 --- f

Leave the Boolean value of an unsigned less-than comparison. Leaves f = 1 for u1 u2; otherwise leaves 0. This function must be used when comparing memory addresses. u1 and u2 are unsigned 16-bit integers.

UNTIL

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

P,C2,LO

Occurs within a colon-definition in the forms:

BEGIN ... UNTIL

At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, execution continues ahead.

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

UPDATE

LO

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

USE

--- addr

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

USER

n ---

LO

A defining word used in the form:

n USER ccc

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

✓ VARIABLE --- E,LO

A defining word used in the form:

n VARIABLE cccc

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

✓ VOC-LINK --- addr U

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

✓ VOCABULARY --- E,L

A defining word used in the form:

VOCABULARY cccc

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

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

✓ VLIST ---

List the names of the definitions in the context vocabulary.

✓ WARNING --- addr U

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

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

Occurs in a colon-definition in the form:

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

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

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

✓ WIDTH --- addr U

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

✓ WORD c --- LO

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

X ---

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

✓ XOR n1 n2 --- xor L1

Leave the bitwise logical exclusive-or of two values.

✓ [--- P,L1

Used in a colon-definition in form:

: xxx [words] more ;

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

✓ [COMPILE] --- P,C

Used in a colon-definition in form:

: xxx [COMPILE] FORTH ;

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

✓] --- L1

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

CREDITS

The following are some of the individuals and organizations that we acknowledge credit for the initial development and advancement of FORTH:

FORTH was created by Charles Moore the founder of FORTH, Inc. We are all indebited to this man and his associates because without their effort this powerful computer tool would not be available.

FORTH INTEREST GROUP (FIG) and Bill Ragisdale developed the fig-FORTH Model that OmniFORTH was derived from. It is recommended that anyone seriously interested in FORTH join the FORTH INTEREST GROUP; an application form is included in this manual. It is also highly recommended that the user order the following manuals from FIG:

fig-FORTH Installation Manual $10.00
Using FORTH, by FORTH, Inc. $25.00

FORTH INTERNATIONAL STANDARDS TEAM (FIST) develops and maintains the standards for which we all try to follow.

FORTH implementation teams (part of FIG) developed the Model on various processors and their results were used as a base in the development of OmniFORTH. Further credits to individuals involved in the implementation effort are presented in the installation manuals for the particular OmniFORTH applications.

INTERACTIVE COMPUTER SYSTEMS, INC.
6403 DIMARCO ROAD • TAMPA, FLORIDA 33614

FORTH INTEREST GROUP

MAIL ORDER

	USA/ CAN	OVERSEAS AIR
☐ Membership in FORTH Interest Group and Volume 2 (6 issues: #7 thru #12 of FORTH DIMENSIONS	$12	$15
☐ fig-FORTH Installation Manual, containing the language model of fig-FORTH, a complete glossary, memory map, and installation instruction	$10	$13
☐ Assembly language source listing of fig-FORTH for specific CPU's. The above manual is required for installation. Check appropriate box(es). Price per each.	$10	$13
☐ 8080 ☐ 6502 ☐ 6800 ☐ PDP-11 ☐ 9900 ☐ PACE		
☐ Volume 1 of FORTH DIMENSIONS. Issues 1 thru 6 as a set.	$6	$8
☐ Reprint of two Dr. Dobbs FORTH articles "FORTH for Microcomputers, "DUMP Example."	$2	$2
☐ Using FORTH, by Forth, Inc. This is the best users manual available. 160 pages, spiral bound.	$25	$31
☐ FORTH Programmers Reference Card. If ordered separately, send stamped, addressed envelope.	FREE	

RENEW NOW! TOTAL $_____

Make check or money order on U.S. bank payable to: FIG.
All prices include postage. No purchase orders.

NAME_____ MAIL STOP/APT_____

ORGANIZATION_____(If company address)

ADDRESS_____

CITY_____ STATE_____ ZIP_____

COUNTRY_____

FORTH INTEREST GROUP · P.O. BOX 1105 · SAN CARLOS, CA 94070