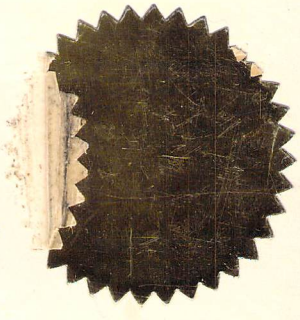
UPDATE +BUF PREV USE M/MOD
*/ */MOD MOD / /MOD * M/
M* MAX MIN DABS ABS D+-
+- S->D COLD ABORT QUIT (
DEFINITIONS FORTH VOCABULARY IMMEDIATE
INTERPRE TACK DLITE L LITE
[COMPI ATE ID. RROR (RT)
-FIND UMBER (NUMB PPE ORD
PAD H D BLANKS E SE
QUER ." (
TYPE CO DOES> <BU ;CODE
(;CODE) DECIMAL HEX SMUDGE]
[COMPILE ?LOADING ?CSP ?PAIRS
?EXEC ?COMP ?ERROR !CSP PFA
N CFA LA TH E
- P SP T -
- LLOT ER 2+ 1+
H CSP P ASE
S TE RRE C T O SET
S OU K DO K P
FENCE WARNING WIDTH TIB +ORIGIN
B/SCR B/BUF LIMIT FIRST C/L
BL 3 2 1 USER VARIABLE
CONSTANT ; : C! ! C@
@ TOGGLE +! DUP SWAP DROP
OVER DMINUS MINUS D+ + 0<
0= R R> >R LEAVE ;S RP!
SP! SP@ XOR OR AND U/ U*
CMOVE CR ?TERMINAL KEY EMIT
ENCLOSE (FIND) DIGIT I (DO)
(+LOOP) (LOOP) 0BRANCH BRANCH

C64 FORTH

***FOR THE COMMODORE-64 COMPUTER**



Performance Micro Products
"The Park at 95"
770 Dedham Street-S2
Canton, Massachusetts 02021

C64-FORTH™

A FORTH LANGUAGE SYSTEM FOR THE COMMODORE-64

by Gregg Harris

C64-FORTH CONFORMS TO:

FORTH-79 STANDARD WITH DOUBLE-NUMBER EXTENSIONS

A product of:

Performance Micro Products
770 Dedham St.
Canton, MA 02021 USA

Any mention within this manual of COMMODORE 64, C64, or 'the 64', is reference to the COMMODORE 64 personal computer manufactured and marketed by COMMODORE BUSINESS MACHINES, INC. The FORTH language standards are defined and maintained by the FORTH INTEREST GROUP (P.O. Box 1105, San Carlos, CA. 94070). This version of C64-FORTH meets the FORTH-79 standard with Double-Number Extensions.

We would like to acknowledge Mr. Charles H. Moore, the creater of FORTH, and all of the contributors of FORTH INC. who have helped make FORTH something more than a word to stick after 'GO'. Some of the descriptions in our ASSEMBLER section reflect the work of Bill Ragsdale, from his treatise on a Forth Assembler for the 6502. The method adopted for scaling in the graphics routines was taken from work by Allen Tracht. Our thanks to Allen for providing a practical implementation.

We believe C64-FORTH to be as close to the FORTH-79 standard as is possible in the word set provided, the way standard words work as generally accepted, and in the manual's attempt to convey the operation of those words. Upon shipment of the first copy, there were no known flaws in achieving the above goal, nor 'bugs' within any of the code. If any user of C64-FORTH finds otherwise, we emphatically encourge notification of such. We would not want buy software from a company that did not care to provide support once the product's out the door, so we're not going to ask you to.

PERFORMANCE MICRO PRODUCTS, INC.
770 Dedham St. -S2
Canton, Mass. 02021

C64-FORTH™

- COPYRIGHT NOTICE

The C64-FORTH package including the object code, and FORTH source screens, both supplied on disk or cassette, and this user manual are serial numbered and are protected by a copyright. Reproduction by any method whatsoever of these copyrighted programs or the user manual is expressly prohibited without the expressed written permission of Performance Micro Products, Inc. To protect your copy of this software, you are permitted to make one backup copy of the contents of this disk or cassette. This backup may not be redistributed.

Except for a single backup copy, it is a Federal crime to make a copy of the manual, floppy disk, or cassette for use by anyone other than the individual who purchased this software, or the individual for which a company purchased this software. A reward will be provided for information which leads to the prosecution and conviction of parties who violate this copyright.

With respect to distribution of application programs created using C64-FORTH, we allow distribution of this software using the "SAVETURNKEY" command with no legal restrictions or license fee, providing that the end user does not have access to the FORTH words contained in the C64-FORTH nucleus. Programs created with "SAVETURNKEY" will operate in a stand-alone configuration and will not require the purchase of a C64-FORTH.

The individual, for whom this software was purchased, may save new or modified versions of C64-FORTH, for his/her own use only, onto disk or cassette by using the "SAVESYSTEM" command under C64-FORTH. Since any program saved under this command is a fully usable FORTH system, the above restrictions on redistribution also apply. Disks or cassettes containing ONLY FORTH source screens, and not any usable copies of any FORTH system generated from C64-FORTH, may be freely distributed.

Serial numbers are embedded in several different formats in several different places in the C64-FORTH software. We intend to identify and prosecute the source of any original or modified copy of C64-FORTH, not generated with the SAVETURNKEY command, that was redistributed for compensation or not.

This policy is in the best interests of the legitimate purchasers and users of C64-FORTH. Performance Micro Products would like to continue to offer a broad spectrum of software products including many extension and applications packages for C64-FORTH. We cannot afford the time and financial demands required to produce very versatile, user-forgiving, and extensively tested software if users of C64-FORTH do not respect our right to compensation for our investment.

- LIMITED WARRANTY INFORMATION

For a period of ninety (90) days from the date of sales, Performance Micro Products, Inc. warrants that the cassette or diskette containing software programs is free from defects. In the event of a defect the customer's sole and exclusive remedy is limited to the correction of the defect by replacement or complete refund at Performance Micro Products election and sole expense. Performance Micro Products has no obligation to replace items that have failed due to abuse or misuse. Peformance Micro Products makes no warranty as to the design, capability, capacity, or suitability for use of the software. Software is licensed to the original retail purchaser on an "as is" basis, without warranty. The customers exclusive remedy in the event of a software defect is the replacement or update of the software within 90 days of the date of purchase upon the return of the original cassette or diskette to Performance Micro Products.

Except as provided herein, Performance Micro Products shall not have any liability with respect to any loss liability or damage caused directly or indirectly by the software sold or furnished by Performance Micro Products. The exclusion of liability shall include but not be limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use of the software.

ABOUT THIS MANUAL

It is not within the scope of this manual to provide a complete introduction to FORTH. There are several good books available that break a new user in on FORTH a little more slowly. The goals of this manual then were defined as:

1) The manual is to serve as a concise reference guide to the words available under, and the architecture of, this C64-FORTH implementation of the FORTH language.

2) The user is assumed to be at least somewhat familiar with the Commodore-64, and has tried writing some programs in BASIC. Equivalent routines in BASIC will be shown whenever a FORTH word or function is similar to one in BASIC.

3) Elaboration on a subject is done to provide enough information so a new user shouldn't need a book on FORTH as long the user is willing to experiment. Elaboration is also done on subjects not usually covered in most books on FORTH.

DON'T BE AFRAID TO EXPERIMENT! Get in there and peek and poke around (or should I say @ and ! around). Look at what you are doing. Take note of conditions before you try something; note results after you try it. Set up conditions to see how different FORTH words work. FORTH can be a lot more rewarding and versatile than languages that 'do it all' for you.

CONVENTIONS IN THIS MANUAL

Any mention of 'the 64', or C64 will of course refer to the Commodore-64, such as "press the STOP key on the 64".

Any input of commands must always be followed by pressing the RETURN key. Nothing will happen until you do. At first, this is indicated in the examples by the symbol <return> as a reminder, but later in the manual it is assumed that you will do it automatically after typing in any commands or numbers.

There are cases where a symbol is used to indicate a string of characters representing a name or something is to be substituted. This will usually be expressed as a keyword within greater-than and less-than symbols. An example is when the symbol ' <name> ' appears, this is an indication that a character string is to be entered, which will be used as the name of a definition being created. If <name> is used later in the description, the SAME character string is to be substituted again.

Examples will have underlined the part typed in by the user ONLY when the response from C64-FORTH is significant. C64-FORTH responds to a line input with 'OK' (or a question mark '?' if there was an error). Later in the manual, the practice of underlining user input and indicating C64-FORTH's 'OK' response is discarded.

Numerical values in examples are sometimes given in DECIMAL, sometimes in HEX. We feel it is a great help to become proficient in the HEX number system as well as with decimal numbers. The relationship or necessity of using a certain value will often be far more obvious when displayed in HEX. The base of the values given in some examples is not stated when it is obvious what it must be.

LOADING AND RUNNING C64-FORTH

Make sure both the 64 and the disk drive are on before inserting the C64-FORTH disk into the disk drive.

Load the program by typing:

LOAD"C64FORTH",8 <return>

and when finished, type:

RUN <return>

A copyright and serial number message should have been printed and the cursor should be flashing. If not, remove the disk, turn power off to both the 64 and the disk drive for a few seconds, and repeat.

MAKING A BACKUP COPY OF C64-FORTH

Upon entering C64-FORTH for the first time, you should immediately make a backup copy of the C64-FORTH disk. With the original C64-FORTH disk still in the drive, type:

2 LOAD <return> (note: the space between '2' and 'LOAD' is important!)

A backup routine will be loaded from disk. Follow all instructions and when done you will have an exact copy of the C64-FORTH disk. You should put the original C64-FORTH disk away, and should use ONLY the backup from now on. Sections in this manual cover how to create an application program disk, how to backup source code screens that you create, and initialize a new screen disk. But until you become more familiar with C64-FORTH, the backup disk you just made is all you need.

CONTENTS

THE FUNDAMENTALS OF FORTH

The FORTH language architecture is built around the DATA STACK. While FORTH allows memory locations to be assigned for use as variables, constants, and work areas, the data stack is used to hold and transfer most values and parameters (DATA ITEMS) for use by the various sections of a program. The data stack allows very quick and convenient access to data items. Since most data items needed by or produced within intermediate portions of a program are temporary in nature, the data stack proves very handy.

The way the data stack works is identical to the way Reverse-Polish-Notation (RPN) type of calculators on the market work. Many analogies have been created to describe the FORTH data stack in use, the following is perhaps one of the closest.

We all have a desk or table somewhere with a pile of stuff on it. When bills come in, we put them on 'top of the stack'. When we get paid, it's time to pay the bills (after all, we are responsible). We take the first bill off the top of the stack and take care of it. We continue until all bills have been removed from the stack and taken care of. A new bill has come in, and there is only one copy of it. We make a photocopy of it and throw that on the pile too so we have a copy for ourselves when we go to pay it.

This basic throw-on-take-off principle is referred to in FORTH as push-on-pop-off. We will refer to data put on the stack as DATA ITEMS. A data item, when added to the stack, is PUSHED ON. A data item, when removed from the stack, is POPPED OFF. In a FORTH program you will usually push appropriate values or data items on the stack, CALL (invoke) a lower-level procedure which will remove or modify values on the stack, and control will return to the higher-level procedure with results left on the stack.

The stack is the mechanism by which portions of a FORTH program communicate with other portions of the program. The FORTH language itself is comprised of a number of commands that operate on memory locations such as variables, constants, and buffers, or on data items on the stack. In other computer languages, these are referred to as instructions. In FORTH, the commands are usually referred to as WORD DEFINITIONS, or more simply WORDS. The number of words making up the FORTH language is variable. The FORTH standards define a minimum subset, and new words can easily be added by the user to make a very versatile and customized higher-level language. The concept of the FORTH word fits well with the concepts of the FORTH vocabularies and dictionary, which will be explained in a minute.

The FORTH program itself is loaded into the COMMODORE-64 just like a BASIC program. FORTH is different however in that a portion of itself is written in FORTH code. FORTH actually is comprised of two types of code. The first is direct machine code routines, just like what BASIC itself is written in. The second type is WORD DEFINITIONS. A WORD definition is comprised of a number of POINTERS. The pointers point to locations that contain addresseses of machine code routines. When a definition EXECUTES, the machine code at the address found at the destination of the first pointer is executed, then the code at the address found at the destination of the second pointer is executed, etc., giving the appearance that the WORD definition is executing.

A word definition has two parts. The HEADER is made up of the definition's NAME, as well as other information. The BODY of the definition contains pointers to addresses of routines that will execute specific functions. Definitions are linked together sequentially in the DICTIONARY. The dictionary may be comprised of several VOCABULARIES. FORTH is the name of the main vocabulary. To see the names of all the words in the FORTH vocabulary, type:

```
FORTH  VLIST   <return>
```

The listing on the screen may be slowed down by pressing the CTRL key on the C64 keyboard, or may be terminated by pressing the STOP key.

In C64-FORTH as shipped, there is only one other vocabulary, the EDITOR vocabulary. To see the words in the EDITOR vocabulary, type:

```
EDITOR  VLIST   <return>
```

Notice that some new names appear, followed by the words in the FORTH vocabulary. All vocabularies link back to FORTH. To return to just the main FORTH vocabulary, type:

```
FORTH     <return>
```

Why are new vocabularies created? You can create new definitions with the same names as words in the FORTH vocabulary. If you create those definitions in the FORTH vocabulary, the original definitions are no longer accessible. If you create them in a new vocabulary, they are. Also, as you enter commands (the names of words), the FORTH system searches through the current vocabulary, then the FORTH vocabulary itself, for a match. If a match is found, the definition is executed. Putting the definitions of seldomly used words into other vocabularies will speed up the searching when just the FORTH vocabulary is selected.

Even though the use of vocabularies suggests a 'tree' structure for the dictionary, it actually is a linear structure. When a new definition is added to the dictionary, it is put onto the end (top) of the dictionary, and information is put into its header that links (logically connects) it to the previously defined word IN THE PRESENT VOCABULARY.

Words are provided to do simple operations on data items on the stack and on the contents of memory locations. You can create and use variables, constants, data arrays, and buffers much the same as in other languages. There is a class of words available called DEFINING WORDS. These words are for creating new WORD TYPES or DATA TYPES. Unlike some other languages, you may create new types of instructions, and very complex custom data storage and manipulation commands. FORTH can be customized to be a very simple, easy to use language, or very complex with extensive high-level capabilities.

FORTH, unlike PASCAL or FORTRAN, is not a strongly 'typed' language. This means that in FORTH, you may freely manipulate values on the stack or in memory considering them as one type of data (e.g. 16-bit integers), then later you may use the data considering it to be another type (e.g. 2-byte pairs of ASCII characters).

In FORTH, unlike most other high-level languages, you have complete access to, and direct control of, the entire computer down to the machine level. This does provide extreme versatility and speed, however there is one major disadvantage - there is no protection of the FORTH system itself. It is quite easy, if one is careless, to overwrite parts of the FORTH system causing a system crash and loss of all recent work. The point here is that until you become an experienced FORTH user, save current work frequently. We don't want to discourage experimentation; we just suggest that you isolate it from useful work.

In other languages, you create new programs by writing source code as one long continuous file. A compiler, such as PASCAL, takes the source code file, reads it in, and converts it to executable machine code (called COMPILING). This machine code is saved as a file, and when the user wants the program to run, he loads the code file and executes it. With interpreters, such as the BASIC in the C64, the source code file is also the executable code in a way. The interpreter scans the program source code file and INTERPRETS (executes machine code routines that performs the function identified by the name of the instruction) each instruction name found. The program source code is one continous file.

FORTH functions in both a compiler and an interpreter mode. Normally, it is in the interpreter mode. When it responds to a line of commands after the RETURN key is pressed, FORTH acts very much like BASIC in the direct mode. FORTH tries to match each word name encountered. The code corresponding to the name is executed as each name is found in the INPUT STREAM. 'Input stream' is another name for the line of commands and/or numeric values that was input in response to the flashing cursor. Only when new words are being created by defining COLON DEFINITIONS (colon defs are explained in the section ADDING NEW DEFINITIONS), does FORTH enter the compiler mode. In this mode, word names are not interpreted as they are being read in from the input stream, but are compiled into memory. This means a new word is being created and added to the dictionary that may be executed later, when back in the interpreter mode, just by typing its name.

The source code for a new program is not written and saved as one continous file however. The concept of SCREENS thus arises. A FORTH screen (not to be confused with the screen of the TV or display that the 64 is hooked up to) is a block of 1024 bytes of data that is transferred to and from the disk or cassette as one unit. When a FORTH screen is LOADED, it is read in from disk, and the FORTH interpreter uses it as the input stream. In effect, it is the same as if you had typed in the 1024 characters in response to the flashing cursor.

By using the editor supplied with C64-FORTH, the user may place into screens the source code to define new word definitions. Since in FORTH the concept of writing a program is to keep definitions short and build up higher-level definitions from lower-level definitions, the 1024 byte limit on a screen is no problem. One screen may itself load another screen. Therefore, one entire program, comprised of source code on many screens, may be loaded with just one command.

Screens may also be used for purposes other than source code definitions. Since the screen, when loaded, is fed into the input stream, any words that aren't part of new colon definitions being created are executed immediately. Thus, a screen may act as an indirect command file. Direct reading and writing of screen blocks to and from memory is possible, allowing use of screens for storing data arrays, tables, machine code subroutines and overlays, etc.

C64-FORTH is shipped with a number of screens on the disk or cassette. Screens 4 and 5 contain ASCII error and status messages, screens 6 through 17 contain the source code for the assembler, and other screens may contain code or data.

ENTERING COMMANDS IN THE FORTH SYSTEM

Before continuing, how to enter commands in the FORTH system should be covered. Whereas in other languages, you have INSTRUCTIONS, OPERANDS (such as an immediate numeric value, name of a variable, etc.), and OPERATORS (such as + and =). In FORTH, you just have WORDS and NUMERIC VALUES. A WORD is an entry in the dictionary. When the name of a word is specified, a piece of code somewhere gets executed. Therefore, to effect a command, you just enter into the input stream the name of a word in the dictionary. You may enter the names of several words at a time in a command (input) line, or just one at a time. Every function available in FORTH has an entry in the dictionary and can be invoked just by entering the name of it.

When you press the RETURN key, all characters on the current line are passed to the FORTH as the input stream. Every word name or sequence of characters representing a number MUST be seperated by AT LEAST ONE SPACE. Since an operator such as multiply (*) in other languages is defined as a word in FORTH, there must be at least one space between numbers and things like + , * , etc. The only way FORTH knows when the name of one word stops. and the name of another word begins, or when a numeric representation begins and ends, is by detection of a space character.

You may be wondering what is the range of characters you may use within the name of a word. The answer is ANY SCREEN-PRINTABLE character may be used within the name of a word. To keep compatability with other FORTH systems however, you should not use any special Commodore graphics characters, only standard ASCII characters when defininig new dictionary entries. The name of any FORTH word may be up to 31 characters in length.

How do you enter comments between FORTH commands (word names) ? Merely by placing the comment within parentheses. But remember, you must place at least one space between the left-parenthesis and the start of the comment, since "(" is a word, a dictionary entry, that when executed, scans off the input stream up to the first right-parenthesis found (No, nesting of comments is not allowed in FORTH).

Why is FORTH called a 'postfix notation' language ? What this gibberish means is that any parameters, qualifiers, numerical values, etc. that a word will need MUST be on the stack BEFORE the word is executed. In BASIC, you specify the instruction first, then the parameters, such a numerical values, the string to print, variable names, etc. Just remember that in FORTH you must specify everything first, then you specify the name of the word you want to execute.

BASIC STACK OPERATIONS

In this manual, the term DATA ITEM is used generically to refer to an item on the stack. A data item in most cases takes up 16-bits (2-bytes) of stack space. Getting a number on the stack is quite simple - just specify it. A string of characters found in the input stream that does not match any entry in the dictionary is converted to a number (if possible) and pushed onto the data stack. The range allowed for number input is -32768 to +65535 (ways of getting around the limits will be discussed later). A number in this range is called a SINGLE PRECISION INTEGER.

Words that operate on items on the stack consider single precision integers to be one of two types:

1) a SINGED SINGLE PRECISION INTEGER. This is a whole number in the range of -32768 to 32767.

2) an UNSIGNED SINGLE PRECISION INTEGER. This is a whole number in the range of 0 to 65535. An example where an unsigned integer will be specified is the address required for fetching data from or storing data to memory or I/O devices.

Note that the unsigned value 65535 is equal to the signed value of -1, 65534 is equal to -2, ... and unsigned 32768 is equal to signed -32768. It is up to the routine that uses or modifies a stack or memory value, as to how the value is interpreted.

To put a number(s) onto the stack, just type in the value(s) in response to the flashing cursor, followed by <return>. An 'OK' is printed. The 'OK' is FORTH's way of showing that the line just input was fully accepted. To print out the top value on the stack, use the "dot" command (the ascii decimal point, or period). Dot removes the top value and prints it as a SIGNED single precision integer.

142 OK

. 142 OK

-7 678 10426 -32000 OK

. -32000 OK

. . 10426 678 OK

. -7 OK

32769 . -32767 OK

Note that the most recently added value to the stack is printed first. At this point, the stack should be empty. To verify, just enter the dot command again.

```
. 0  EMPTY STACK
```

If the empty stack message didn't occur, you somehow had extra values on the stack. Keep executing the dot command until the message occurs. The value printed just before the empty stack message is always meaningless, since the check for the stack being empty is always made upon return back into the input prompt routine.

The word used to print the top stack value as an UNSIGNED integer is U. (pronounced "u dot").

```
-1 U.  65535  OK
```

- MANIPULATING STACK ITEMS

There are several FORTH words available for duplicating, removing, or changing the position of stack items. To assign an identity to different stack items for the purpose of showing effects on stack items by FORTH words, a certain notation is used. This is described more fully on page 1 of appendix E, the start of the glossary. Basically, n1 means number #1, n2 means number #2, etc. If n2 is to the right of n1, then n2 is higher on the stack (closer to the top) than n1. The three dashes ("--- ") indicate the execution point of the word. Values to the left of the dashes are what's required on the stack before the word is executed, values to the right are the results after execution of the word.

Mention should be made here as to the function of the GLOSSARY at the back of this manual. The glossary works like a normal dictionary. The names of all the words in the base FORTH vocabulary are listed, with descriptions of how the words function, and their stack requirements and effects. Once you get up to speed in FORTH, you will probably be using only the glossary part of this manual. There are seperate glossaries for words available for graphics use, strings, floating-point, and under the ASSEMBLER. These have been placed in seperate sections of the manual since they are not necessary in normal FORTH programming.

The stack manipulation commands are:

DUP n1 --- n1 n1
duplicates the top stack item

SWAP n1 n2 --- n2 n1
exchanges the top two stack items

DROP n1 ---
removes and discards the top stack item

OVER n1 n2 --- n1 n2 n1
copies the second item and pushes it on the stack

ROT n1 n2 n3 --- n2 n3 n1
rotate the top three stack items

n PICK ... n4 n3 n2 n1 n --- ... n4 n3 n2 n1 n?
copy the n-th stack item (not counting n itself) and push it onto the stack. e.g. 1 PICK is the same as DUP, 2 PICK is the same as OVER.

n ROLL ... n4 n3 n2 n1 n --- ... n? n? n? n?
rotate the top n stack items (not counting n itself) e.g. 3 ROLL is the same as ROT, and 4 ROLL has the effect: n4 n3 n2 n1 4 --- n3 n2 n1 n4

Examples:

37 DUP . . 37 37 OK

123 19 SWAP . . 123 19 OK

42 12 DROP . . 42 0 EMPTY STACK

78 6 OVER . . . 78 6 78 OK

10 20 30 ROT . . . 10 30 20 OK

10 20 30 40 4 PICK OK

. . . . . 10 40 30 20 10 OK

10 20 30 40 4 ROLL OK

. . . . 10 40 30 20 OK

- ARITHMETIC OPERATIONS

It is very simple to perform arithmetic operations on values on the stack. All values necessary must be on the stack before the operation is performed:

```
3 17 + .  20  OK

20 32 - .  -12  OK

7 8 * .  56  OK

204 17 / .  12  OK
```

FORTH is a 'free-format' language. You may specify one operation per line or many operations per line. You may also control when operations are done, as long as the required number of values are on the stack:

```
3 7 2 8 + + + .  20  OK

3 7 + 2 + 8 + .  20  OK

3 7 + 2  OK

8 + + .  20  OK
```

Operations are performed in the order that they are specified:

```
3 7 8 + * .  45  OK              ( equiv. to (7+8)*3 )

3 7 8 * + .  59  OK              ( equiv. to (7*8)+3 )
```

There are several other words in FORTH for arithmetic operations. Refer to the glossary for descriptions on:

```
*/    */MOD    /MOD    MOD
1+    1-    2+    2-    2*    2/
ABS    MAX    MIN    NEGATE
```

- WORKING IN DIFFERENT NUMBER BASES

Numbers in base 10 (the decimal system) are not at home in computers. The relationship between addresses, bit patterns, displays of memory contents, etc. become more apparent when working in other number bases. While decimal numbers are sufficient in BASIC, for languages such as FORTH or assembly that allow you more absolute control over the computer hardware, HEXadecimal (base 16) is more meaningful.

BASE is a FORTH variable that keeps track of the current number base. All input and output number conversions (characters into binary values and visa versa) are done according to the current base.

DECIMAL sets BASE to 10. All numbers input are expected to be decimal numbers, and all numbers printed out are decimal.

```
DECIMAL 8 9 + .  17  OK
```

HEX sets BASE to 16:

```
HEX 8 9 + .  11  OK

1A C + .  26  OK
```

The current BASE selected stays in effect until changed. If you were brought-up in mini-computer land, you are probably used to OCTAL numbers. You may work with octal numbers by setting BASE to 8:

```
8 BASE ! 7 6 + .  15  OK
```

Other bases (such as BINARY) may be selected in a similar way (2 BASE !).

- CONSTANTS AND VARIABLES

In BASIC, a constant is a variable whose value does not change throughout the course of the program. In FORTH, CONSTANTS and VARIABLES are uniquely different data types.

Constants and variables are dictionary entries, just like word definitions. When the name of a constant is executed, the value of that constant is pushed onto the stack. The numbers 0, 1, 2, and 3, because they are used so frequently, are actually defined as constants. Since the dictionary is searched before a numeric conversion is attempted, this speeds things up.

There are other constants in FORTH, such as C/L. C/L, when executed, pushes onto the stack the number of characters per display line:

```
DECIMAL  C/L  .  40  OK
```

The word CONSTANT itself is a defining word. It is used to create new constants:

<initial value> CONSTANT <constant name>

Example:

```
7  CONSTANT  DAYS/WK  OK
```

creates a new dictionary entry, called DAYS/WK , and assigned the value of 7 to it. Whenever DAYS/WK is executed, the value 7 will be pushed onto the stack:

```
DAYS/WK  .  7  OK
```

VARIABLES are used extensively in FORTH. As in BASIC, variables may be initialized and changed easily. When the name of a variable is executed, the ADDRESS (location) of the contents of the variable is pushed onto the stack.

In the dictionary, a constant or variable is made up of a header (containing the name and linkage and execution-code pointers), and a body consisting of two bytes. The 16-bit value of the constant or variable is stored in these two bytes. The only difference is that a constant will fetch the contents of the body and put it on the stack, and a variable will put the address of the body (location of the value) on the stack.

The word VARIABLE itself is a defining word. It is used to create new variables:

VARIABLE <variable name>

Example:

```
VARIABLE #OFERRORS  OK
```

When #OFERRORS is executed, it leaves the address of its current value. When a variable is created, its initial value is random. To initialize it, a value must be stored in it as described in the following section.

In FORTH, there are a number of USER VARIABLES (e.g. BASE). The contents of the variable are not stored immediately following the header in memory, but in a table called the USER AREA located at the top of memory used by the FORTH system. Although functionally identical to variables, executing the name of a user variable leaves the address within the user area of the contents of the variable.

- FETCHING AND STORING

Two words, @ ("fetch") and ! ("store") are used to read from, or write to variables or memory locations. @ 'fetches' (reads the 16-bit contents of) the memory location pointed to by the address on the stack. The address on the stack is replaced by the contents read from that location:

```
641 @ .  2048  OK
```

The equivalent in BASIC is:

```
PRINT(PEEK(642)*256+PEEK(641))
```

Since executing the name of a variable leaves an address on the stack, then @ can be used to read the contents of a variable.

From a previous example:

#0FERR0RS @ . 0 OK

(the value printed may not be 0 since we did not initialize the variable #0FERR0RS yet.)

BASIC:

200 PRINT NE

User variables may be read the same way:

BASE @ . 10 OK

! (store) is used to write a value to a 16-bit memory location. ! requires two values on the stack. The second stack value must be the value to store, and the top stack value must be the address to store it at. Both values are removed from the satck after the store is done.

513 1142 ! 0 55414 ! OK

The letters "AB" should have appeared in black in the upper right corner of the display screen. Since executing the name of a variable leaves an address on the stack, then ! can be used to change the contents of a variable:

10 BASE ! OK

- FETCHING AND STORING BYTES

@ and ! are used for 16-bit values. Since the 6502/10 is an 8-bit machine, words are provided for reading from and writing to byte locations. C@ is used for 8-bit fetching, and C! is used for 8-bit storing.

(what is the current border color ?)

53280 C@ . 14 OK

(change the current border color)

4 53280 C! OK

(restore the border color to start-up value)
(Note: the variable BORDER contains the color value that C64-FORTH sets the border to upon start-up)

BORDER C@ 53280 C! OK

- DOUBLE PRECISION NUMBERS

Since each position on the stack can hold a data-item consisting of 16-bits, the number range is limited to -32768 to 65535. This is rather limited for some types of applications, so a class of words is provided in FORTH to extend the range of possible stack values. Called DOUBLE PRECISION INTEGERS, the effective range may be -2,147,483,648 to 2,147,483,647. Only SIGNED double precision integers are supported in this version of C64-FORTH. Double numbers take up 32-bits (4-bytes) of stack space.

A double number is input by placing a decimal point somewhere within the number. No matter where the decimal point is placed, the same value ends up on the stack, but the location where the decimal point was found is placed in the user variable DPL.

The word for removing the double number on top of the stack and printing it as a SIGNED double integer is D. ("d dot"):

```
1234.567 D.  1234567  OK

DPL @ .  3  OK

12.34567 D.  1234567  OK

DPL @ .  5  OK

123456.7 D.  1234567  OK

DPL @ .  1  OK
```

A double number on the stack may be treated as two single integers. The high order part of the number (the high 16-bits) is the top stack item, the low order part is the second stack item:

```
HEX 12345678.  OK

.  1234  OK

.  5678  OK
```

To convert a positive single integer on the stack to a double integer, just push a zero. To convert a negative single integer on the stack to a negative double integer, just push 65535. The word S->D (for 'single-to-double') automatically converts a single precision integer into a double integer, accounting for the sign of the integer. To convert a double integer into a signed single integer, just execute DROP to remove the top 16-bits of the double number.

Double number equivalents to most of the single integer stack manipulation commands exist. They are the same commands preceeded by a '2'.

```
2!   2@
2DUP   2DROP   2SWAP   2OVER   2ROT
```

2CONSTANT and 2VARIABLE work similar to their single number counterparts:

```
DECIMAL  8732109.  2CONSTANT  BIGNUM  OK

BIGNUM  D.  8732109  OK

2VARIABLE  BNUM  OK

99998888.  BNUM  2!  OK

BNUM  2@  D.  99998888  OK
```

Double precision arithmetic operations usually begin with a 'D'.

```
D+   D-  DABS   DMAX   DMIN   DNEGATE
```

There are several words used that accept or produce both single and double precision numbers. Refer to the glossary for their functions:

```
D+-   M*   M/   M/MOD   S->D   U*   U/
```

FLAGS AND COMPARISONS

- FLAGS

FORTH uses flags extensively. Actually, a flag placed on the stack is just a numeric value. If the value placed on the stack is a zero (0), it is said to be a FALSE flag. If a value placed on the stack is non-zero, it is said to be a TRUE flag. A true flag is usually a value equal to 1, but not always. Many FORTH routines return a condition, either true or false, to indicate whether something did or did not happen, something was or was not within a specified range, etc. Program branching and looping directives, described later, may test the value of the flag, and take action depending on its state.

- COMPARISONS

Several FORTH words are provided for doing comparisons between numeric values on the stack. A flag is returned by the comparison routine. In the case of 0= , a true flag is left on the stack if the top stack value when the routine 0= was called was equal to zero (0). 0= removed the top stack value in the process of testing it. If the value was not equal to zero, the flag returned is false, or equal to 0. 0= can thus be seen to be useful for inverting the TRUTH of a flag, i.e. using 0= on a flag on top of the stack will return a FALSE value of the flag was TRUE, and will return a TRUE value if the flag was FALSE. (NOT is another FORTH word that is identical to 0=)

There are two other comparators that test just one stack value. They are 0< which tests if the top stack value is less than 0, and 0> which tests if the top stack value is greater than zero. Both of these assume the top stack item is a signed single integer.

```
1 0= .  0  OK

0 0= .  1  OK

-2 0< .  1  OK

28 0> .  1  OK

65530 0> .  0  OK
```

There are four comparators that operate on the top two stack values: < = > and U< . < and > assume the two values are signed single integers, U< assumes they are unsigned single integers, and = just plain doesn't care.

The relationship tested between comparators that require two stack items is:

<second stack value> COMPARISON <top stack value>

Thus, 3 18 > will return a FALSE flag since 3 is not greater than 18.

There are five comparators for testing the relationship of signed and unsigned double precision integers:

D0= D< D= D> DU<

Refer to the glossary for descriptions.

- DO LOOPS

A DO loop is similar to the FOR..NEXT loop in BASIC. A loop is set up with a starting value, an ending value, and an increment. The value that changes with each pass through the loop is called the INDEX of the loop, and the value that serves to terminate (cause the exiting of) the loop is called the LIMIT.

A DO loop is constructed as follows:

```
...  <limit+1>  <starting value>  DO  .. loop code ..  LOOP
```

A value that is equal to the desired loop limit +1 is pushed onto the stack, the value representing the starting value of the index is pushed onto the stack, then the loop is entered. It is important to understand that the words DO and LOOP can only be used inside of a colon definition, because at compile-time they create the structure that at execute-time will provide a looping effect!

The limit must be specified as 1 greater than the value desired to terminate the loop since the index is incremented BEFORE it is compared against the limit.

As an example, lets print the first 5 characters of the alphabet. Since a DO loop can only be implemented within a colon definition, we must define a word with the loop and then execute the word:

```
: ALPHA5  70  65  DO  I  EMIT  LOOP  ;

ALPHA5  ABCDE  OK
```

The FORTH word I fetches the current index of the loop and pushes it onto the stack. The equivalent of the above routine in BASIC is:

```
10 FORI=65TO69
20 PRINTCHR$(I);
30 NEXTI
```

DO loops may be nested:

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

To obtain the current index of the next outer loop the FORTH word J is used, and to obtain the index of the third outer loop K is used:

```
:  LOOPTEST  3  0  DO
                2  0  DO
                  CR  ." I="  I  .  ." J="  J  .
                LOOP
             LOOP  ;
```

```
LOOPTEST
I=0 J=0
I=1 J=0
I=0 J=1
I=1 J=1
I=0 J=2
I=1 J=2  OK
```

In BASIC, the above routine would be:

```
10 FORJ=0TO2
20 FORI=0TO1
30 PRINT"I=";I;"J=";J
40 NEXTI
50 NEXTJ
```

LOOP sets up the structure of the definition being created such that at execute-time a value of +1 is added to the current index, then the index is compared with the limit. A different increment, including negative values, may be specified by using +LOOP instead of LOOP. +LOOP takes the top stack value and adds that to the loop index. The loop index is then compared against the limit, and if it exceeds the limit, the loop is exited:

```
:  MINUSSTEP  -1  100  DO  CR  I  .  -10  +LOOP  ;
```

is equivalent to:

```
10 FOR I= 100 TO 0 STEP-10
20 PRINTI
30 NEXTI
```

NOTE: both LOOP and +LOOP work according to SIGNED arithmetic. This means that the loop will be terminated either by the current index exceeding the limit, OR arithmetic overflow occuring when COMPARING THE CURRENT INDEX TO THE LIMIT. For instance, if the limit of the loop was specified as a negative number, and the current index is being incremented in the positive direction, the loop will terminate after only one iteration. Similarly, if the limit was specified as positive, and the current index is increasing in the negative direction, the loop will terminate after only one iteration.

A DO loop may be prematurely terminated by executing LEAVE. LEAVE sets the current index equal to the loop limit. However, the DO loop IS NOT EXITED UNTIL THE LOOP POINT IS ENCOUNTERED. Since desire to terminate a DO loop prematurely is probably based upon some event happening, LEAVE can be executed within the IF..ELSE..THEN structure that detected the condition:

```
DO  ..  <flag>  IF  LEAVE  ELSE  ..loop code..  THEN  LOOP
```

Since LOOP and +LOOP work according to SIGNED arithmetic, this means that absolute memory addresses cannot be used for the index and limit of a loop if the address range crosses over the boundary at 32768 (8000 hex). There is a word that has been PROPOSED as a future FORTH standard word called /LOOP . /LOOP works like +LOOP but according to UNSIGNED arithmetic, thus absolute memory addresses may be used. The current FORTH PROPOSAL states that /LOOP is to work only with the index being incremented in the POSITIVE direction. /LOOP as provided in this implementation of C64-FORTH works in BOTH directions to allow you to increment the index in both positive AND negative directions.

Examples of usage (all numbers are in HEX):

```
... 8020  7FF0  DO  I  C@  FF  XOR  I  C!  3  /LOOP  ...
```

will complement every third byte in the range of 7FF0 to 801F

```
...  77FF  8FFF  DO  0  I  C!  -1  /LOOP  ...
```

will clear all bytes in the range 8FFF down to 7800 starting with the highest byte first.

- IF...ELSE...THEN

Since FORTH does not allow line numbers or labels, structures have been provided for program branching. IF..ELSE..THEN is used as follows:

```
<flag>  IF  true part  ELSE  false part  THEN
```

The IF..ELSE..THEN structure can only be used within a colon definition. At execute-time, a flag is expected on the stack upon entering the structure. The code that was compiled into the definition by IF tests the flag. If it represents a TRUE condition, then the code between IF and ELSE (the true part) is executed. If the flag is FALSE, then the code between ELSE and THEN (the false part) is executed. Either the true part OR the false part is executed, not both. Whichever part was entered, execution continues after THEN.

```
:  TESTFLAG  ( print whether flag on stack is true or false )
     IF  ." FLAG IS TRUE"
     ELSE  ." FLAG IS FALSE"  THEN  ;

0  TESTFLAG  FLAG IS FALSE  OK

1  TESTFLAG  FLAG IS TRUE  OK
```

BASIC equiv:

```
10 IF C=0 THEN 40
20 PRINT"C IS TRUE"
30 GOTO 50
40 PRINT"C IS FALSE"
50 REM ** REST OF PROGRAM **
```

IF..THEN is a special case allowing execution of a section of code only if a flag on the stack tests TRUE. If FALSE, execution just skips over the IF..THEN part.

```
:  TEST1  IF  ." FLAG IS TRUE "  THEN  ." THIS IS COMMOM CODE"  ;

0  TEST1  THIS IS COMMON CODE  OK

1  TEST1  FLAG IS TRUE THIS IS COMMON CODE  OK
```

BASIC equiv:

```
10 IF C=0 THEN 30
20 PRINT"C IS TRUE"
30 PRINT"THIS IS COMMON CODE"
```

- BEGIN..UNTIL

BEGIN..UNTIL is handy for looping until a certain condition exists. The condition (flag) is tested just before the UNTIL part:

```
BEGIN  loop code  <flag>  UNTIL
```

The loop code is always executed at least once. Some comparison or operation is done at the end of the loop code that leaves a flag on the stack for the UNTIL part to check. If the flag tests FALSE, execution loops back to just after the BEGIN part. If the flag tests TRUE, execution falls through to after the UNTIL part.

```
( wait until the 'Y' key is pressed )
:  BTEST  BEGIN                    ( start of loop )
            GET                    ( read keyboard )
            89  =  UNTIL  ;       ( Y key = value of 89 )
```

BASIC equiv:

```
10 GETA$
20 IF A$ <> "Y" THEN 10
```

- BEGIN..AGAIN

BEGIN..AGAIN creates what amounts to an infinite loop:

```
BEGIN  ..loop code..  AGAIN
```

The loop code is executed forever. While this may not immediately seem useful, certain process control or turnkey system applications will need this structure.

- BEGIN..WHILE..REPEAT

BEGIN..WHILE..REPEAT forms a structure that is an infinite loop, but can be exited upon a certain condition:

```
BEGIN  ..part1..  <flag>  WHILE  ..part2..  REPEAT
```

When the BEGIN..WHILE..REPEAT structure is entered, part1, if it exists, is executed. A flag placed on the stack by a comparison or operation is tested. If the flag is TRUE, the following code (part2) is executed. If the flag tests FALSE, the loop is exited and execution continues after the REPEAT. If part2 is executed, then control jumps back to just after the BEGIN.

Once again, any control structure (DO..LOOP, IF..THEN..ELSE, and BEGIN.. structures) can only be used within a colon definition! Any attempt to use them while in the interpretive mode will generate an error.

- NESTING OF LOOPING STRUCTURES

Nesting of looping and branching structures is allowed as long as EACH STRUCTURE IS FULLY CONTAINED WITHIN ANOTHER. For instance, an IF..THEN structure cannot be used to branch out of a DO loop or BEGIN..UNTIL structure:

```
DO  ..  IF  ..  LOOP  THEN                    NOT ALLOWED!
BEGIN  ..  IF  ..  UNTIL  THEN                NOT ALLOWED!
IF  ..  BEGIN  ..  THEN  .. UNTIL             NOT ALLOWED!
```

These are examples of allowable structures:

```
BEGIN  ..  IF  ..  ELSE  ..  THEN  ..  UNTIL
DO  ..  BEGIN  ..  WHILE  ..  REPEAT  ..  LOOP
IF  ..  DO  ..  BEGIN  ..  UNTIL  ..  LOOP  ..  THEN
```

- SIMPLE ARRAYS

Looking at how FORTH works, we see three things that must be done to create and use arrays:

1) The amount of space necessary must be reserved in the dictionary so FORTH does not overwrite the contents of the array

2) A header and name must be given to the array so we may obtain the address that the data starts at

3) Initializing or changing array entries must be easy

CREATE is a FORTH word used to create a header, assign it the following name, and link it to a code routine that when executed, leaves the address of the first byte in the body. Used as follows:

CREATE <defname>

A new definition, called <defname>, is added onto the end of the dictionary, and is linked to the current vocabulary. There is no space allocated for the body of <defname>. When <defname> is later executed, the address of the first byte after its header is pushed onto the stack.

Space may be allocated with the ALLOT command. ALLOT takes a specified number, and adds it to the end-of-the-dictionary pointer. The contents of the space that is allocated is not initialized or altered in any way.

Instead of ALLOT, we may COMPILE specific values into memory. This process places an 8 or 16-bit value onto the end of the dictionary, and advances the end-of-the-dictionary pointer by 1 or 2. C, compiles a byte value into memory advancing the dictionary pointer by 1, and , (comma) compiles a 16-bit value into memory advancing the dictionary pointer by 2. With C, and , we now have a way to allocate and initialize memory at the same time.

Suppose we would like to create a table of squares for the first ten integers. In BASIC we would have:

```
10 DIMA(10)
20 FORX=0TO9
30 READN:A(X)=N
40 NEXTX
50 REM *** REST OF PROGRAM ***
1000 DATA 1,4,9,16,25,36,49,64,81,100
```

In FORTH:

```
CREATE SQUARES
1 C, 4 C, 9 C, 16 C, 25 C, 36 C, 49 C, 64 C, 81 C, 100 C,
```

A table of 10 bytes is now created. When SQUARES executes, the address of the first byte within the table is pushed onto the stack. Offsets may be added to this value to point to other values in the table:

```
SQUARES 2+ C@ .  9  OK

SQUARES 9 + C@ .  100  OK

150 SQUARES 8 + C!  OK

SQUARES 8 + C@ .  150  OK
```

Tables of 16-bit values may be created with , (comma) instead of C, . If the table need not be initialized, but will be filled by the program, just use ALLOT:

```
CREATE  SQUARES  10  ALLOT
```

- CREATE and DOES> structure

CONSTANT, VARIABLE, and ":" have been referred to as defining words, meaning that they are used to create new definitions in the dictionary. Most FORTH programmers can go the rest of their lives using just these three types for creating all new definitions. This explanation of the CREATE and DOES> structure can be skipped over by new FORTH users to avoid confusion.

DOES> is always used with CREATE within a colon definition. The CREATE - DOES> team is used to create new defining words (similiar to defining a new data type in some other higher level languages, such as PASCAL, but in FORTH it is really generating a new WORD TYPE). The greatest use for creating new word types with the CREATE - DOES> structure is for defining arrays.

CREATE and DOES> are always used together in the same definition. There are two seperate operations to consider about any defining word. The first is what happens when the defining word is used to create a new definition. The second is what happens when the newly-defined word executes.

CREATE - DOES> allows you to precisely define both circumstances. They are used as follows:

```
: <typename>  CREATE  ..part1..  DOES>  ..part2..  ;
```

A new defining word is created called <typename>. The new defining word can be used to create new word definitions or data types:

```
<typename>  <defname>
```

When <typename> is executed, a new definition is created with the name <defname>. When building the definition <defname>, the FORTH code between the CREATE and DOES> part of <typename> (..part1..), is executed. Typically, this code will allocate space for an array, and provide a way to initialize values within the array if desired.

When <defname> is later executed, the address of the first byte in the body of <defname> is pushed onto the data stack, THEN the FORTH code between the DOES> and ; part of <typename> (..part2..) is executed. Typically, this will provide the address of a specific entry within the table, provide the entry within the table, or somthing similar.

As an example, even though the defining word CONSTANT is already present within the FORTH vocabulary, we can redefine it:

```
: CONSTANT  CREATE  ,  DOES>  @  ;
```

When CONSTANT is used to define a constant:

```
12  CONSTANT  MON/YR
```

a new header is created in the dictionary with the name MON/YR. Then the part between CREATE and DOES> is executed. The comma takes the top value off the data stack, and compiles it into memory. The header called MON/YR now has a 16-bit value equal to 12 in the first two bytes after it.

When MON/YR is later executed, the address of the first byte of its body is pushed onto the stack. Then the part between DOES> and ; of CONSTANT is executed, which fetches the 16-bit contents of the location pointed by the top stack item, i.e. the address just pushed. Likewise, we can see that the defining word VARIABLE would be defined as:

```
: VARIABLE CREATE 0 , DOES> ;
```

Here the part between DOES> and ; is nonexistent, so nothing is done to the address left on the stack.

Now, to redefine our previous array SQUARES. We want to create a defining word that can be used to create arrays. We want it to be capable of creating and initializing an array of 10 entries in one case, 18 entries in another case, etc.. When we later refer to the name of a array created, we want it to return the address of a specific entry within the array.

Lets create our new defining word:

```
: ARRAY CREATE 0 DO C, LOOP DOES> + 1- ;
```

ARRAY is then used as follows:

```
100 81 64 49 36 25 16 9 4 1 10 ARRAY SQUARES
```

A new array called SQUARES is created. The first thing that happens after the header is formed, is the top stack value is taken as the limit of the DO loop. This should be the number of entries that will be in the array being created. Next, the DO loop is entered. For as many times as there are entries in the array, a value will be pulled off the data stack and compiled into the dictionary. This will effectively both allocate space for and initialize each entry. In this example, first the 1 is pulled off the stack and placed in the first entry, then a 4 in the second, etc.

When SQUARES is later executed in the following manner:

```
6 SQUARES
```

the address of the first entry of the table is pushed onto the stack. The part between DOES> and ; of ARRAYS is then executed, which adds the address to the previous value on the stack, in this case the 6. Since we are expecting the address of the 6th entry in the table, and entry 1 is in the address+0 position, a 1 is next subtracted from the effective address. The resulting address is the location of the 6th entry in the table. It may be accessed by:

```
6 SQUARES C@ . 36 OK
```

Other arrays may be created with the same defining word:

```
31 30 31 30 31 31 30 31 30 31 28 31 12 ARRAY DAYS/MONTH
```

The number of days in April is found as:

```
4 DAYS/MONTH C@ . 30 OK
```

To define ARRAY for creating arrays that don't need initialization:

```
: ARRAY CREATE ALLOT DOES> + 1- ;
```

and is used in the form:

```
10 ARRAY TEMPVALS
```

With techniques explained later, large arrays can be initialized from data stored in screens.

- COLON DEFINITIONS

The most common way to add new commands to the FORTH system is to create COLON DEFINITIONS. Most of the words in the base FORTH dictionary are defined as colon defs. A colon definition is essentially a list of pointers to previously-defined words. When FORTH appears to 'execute' a word that was defined as a colon def, actually it is executing the code (or colon def) identified by each pointer in the body of the definition.

A colon definition, as the name implies, starts off with a colon (':'). Next, the user supplies the name which is to be assigned to it. The names of previously-defined FORTH words make up the body of the definition, which is terminated with a semi-colon (';'):

```
:   <name>   . . . <body> . . .   ;
```

Example:

```
:   CLRSCREEN   147   EMIT   ;
```

A new word, named CLRSCREEN, has been added to the FORTH dictionary. To see this, type VLIST . CLRSCREEN is now the top word in the current vocabulary.

When 'CLRSCREEN' is typed in, the body of the definition will be scanned and executed. First, the value 147 is pushed onto the stack, then the routine EMIT is executed. EMIT takes the top value off the stack and prints it as a character (printing a value of decimal 147 happens to clear the screen).

Notice that when you were defining the new definition CLRSCREEN, the screen did not clear. Normally, if you type 147 EMIT in response to the flashing cursor, the screen clears. However, when a colon is encountered in the input stream, the FORTH system stops executing commands (interpreter mode) and starts COMPILING a definition (i.e. enters the compiler mode). A HEADER entry is created in the current vocabulary, the following <name> is assigned to it, linkage pointers are added, and all subsequent numeric values and previously-defined FORTH words up to the semi-colon are COMPILED into the definition. 'COMPILED into the definition' means a pointer is added onto the body of the current definition that either points to another definition to execute, or to a routine that at execute-time will push the specified numeric value onto the stack. Encountering the semi-colon turns off the compile mode and FORTH re-enters the interpreter mode. Any word names now found in the input stream will be executed immediately again.

Colon defs may be NESTED to any depth. This means once a definition has been created, it may be referenced in the body of a new definition, which may be referenced in the body of an even newer definition, etc. For example, having already defined CLRSCREEN, we may enter:

```
: 3RDLINE  CLRSCREEN  CR  CR  ;
```

When 3RDLINE is executed, the screen is cleared and the cursor is moved down to the third screen line. 3RDLINE, now defined, may be used in the definition of a new command, and so forth:

```
: SHOWNEST  3RDLINE  ." EXAMPLE OF NESTING"  ;
```

Most of the words in the FORTH glossary may be used inside of a colon definition. There are some words classified as IMMEDIATE. These words overide compile mode and execute immediately. Examples are the names of vocabularies (ASSEMBLER, EDITOR, etc.) and branching control structures (IF..ELSE..THEN, BEGIN..UNTIL, etc.). Words marked as IMMEDIATE are not compiled into the current definition, but perform a function necessary during the compile mode. Good examples of colon defs are the definitions of some of the FORTH words:

```
: DECIMAL  10  BASE  !  ;
: HEX  16  BASE  !  ;
: PAD  HERE  68  +  ;
: BLANKS  BL  FILL  ;
: DEFINITIONS  CONTEXT  @  CURRENT  !  ;
: UPDATE  PREV  @  @  32768  OR  PREV  @  !  ;
```

How many FORTH words can you use to make up the body of a new definition? As many as you want. It is customary in FORTH programming to keep definitions short, but it is not a rule. There is a limit when typing in an input line. 80 characters is the maximum the 64 allows you to input at one time. However, since FORTH is free-format, you may create a definition by inputting many lines. When entering a new definition longer than an 80-character input line, you'll notice that FORTH does not respond with the familiar 'OK' response when the RETURN key is hit. When you finally terminate the new definition by using the semi-colon, FORTH re-enters the interpreter mode and responds with 'OK'.

When you type in a colon definition as an input line, you loose the 'source code'. There is no easy way to examine a word definition to determine the sequence of words that was used to define it. Therefore, it is good practice to enter all definitions, constant and variable declarations, defining arrays, etc. onto source screens. The LOAD command reads in a source screen treating it as if it were typed in as one long input line. The advantage gained here is that it can be permanently stored onto disk or tape, aiding future work. The LOAD command is used as follows:

```
<scr#>  LOAD
```

The screen# <scr#> is read in from disk if not already in memory, and the FORTH input interpreter begins at the start of the screen considering it as a 1024 byte long input line. Normally, when the 1024 bytes are all read in, FORTH prints 'OK' and returns to expecting input from the keyboard buffer. If the word '-->' is found in the screen being LOADed, then the FORTH interpreter stops using the current screen, looks for the next screen in memory or on disk, then feeds that screen into the input stream. If an error is encountered anywhere, LOADing of the current screen immediately stops and an error message is printed. At this point, and before entering ANY other command, if you type WHERE <return>, FORTH will display the current line within the current screen the error occurred at, and with an up-arrow will point to just after the word/character string the error occured in.

Say you had a line in screen #20 that read:

```
:  TEST  CR  XXX  ;
```

and you do not have a word already defined in the dictionary called XXX. When you go to load the screen you will get an error, and when you type WHERE you'll see the following:

```
20  LOAD  20  XXX  ?  CAN'T FIND
WHERE
  :  TEST  CR  XXX  ;
                    ↑
```

Right after entering the LOAD command, FORTH echoes the screen number it is LOADing. When XXX was encountered, no entry was found in the dictionary, so it is printed out with the ? CAN'T FIND error message. When WHERE was typed in, FORTH printed the current line that contained the error, and on the following line printed an up-arrow just after the word that was not recognized.

Many of the screens on the C64-FORTH disk contain source code written in FORTH. Analysis of the routines plus experimentation is the best way to learn how to define new definitions. Create some of your own source screens using the editor (the editor is described in Appendix A) and try loading them.

- SINGLE CHARACTER OUTPUT

A value on the stack may be printed as an ASCII character by using the word EMIT. The value to be printed should be in the range 0 to 255.

```
65 EMIT 67 EMIT AC OK
```

BASIC equiv:

```
PRINTCHR$(65);CHR$(67);
```

- TEXT OUTPUT

There is an easy way to print out ASCII messages. ." (dot-quote) is a FORTH word used to print out a string of characters. It is used as follows:

```
." <text string>"
```

There must be one space between ." and the start of the message. The text message is terminated by a quote character. ." prints out the character string starting one space after it up to the terminating quote character.

Example:

```
." TEXT STRING OUTPUT TEST" TEXT STRING OUTPUT TEST OK
```

BASIC equivalent:

```
PRINT"TEXT STRING OUTPUT TEST";
```

." may be used within a colon definition to print out a message on the screen when the definition is executed:

```
: ASCNUMS CR ." 0123456789" ; OK

ASCNUMS
0123456789 OK
```

CR, SPACE, and SPACES are specialty words. CR outputs a 'return' and is the same as executing PRINT by itself in BASIC. To print 3 returns in FORTH:

```
CR  CR  CR

 OK
```

In BASIC:

```
PRINT:PRINT:PRINT
```

SPACES prints a specified number of spaces on the screen:

```
9  SPACES            OK
```

SPACE prints just one space.

```
." ABC"  SPACE  ." DEF"  ABC DEF  OK
```

FORTH stores character strings in memory as a string of character bytes preceeded by a byte count. To print out a character string in memory in this format, the FORTH word TYPE is used. The address of the start of the character bytes is specified, along with the byte count. Let's place a short string into memory. We'll use EBUF, which is a 30-byte buffer in FORTH used to hold the status message sent back from the disk.

```
( place count of characters followed by character bytes in EBUF )

3 EBUF C! 65 EBUF 1+ C! 49 EBUF 2+ C! 33 EBUF 3 + C!  OK

EBUF  1+  3  TYPE  A1!  OK
```

Why did we store the byte count in the first byte when we didn't need it? To demonstrate another word, COUNT, which is given an address of the start of a FORTH string in memory. COUNT returns the starting address of the character bytes, followed by the byte count taken out of the head of the string. Since this is just what TYPE wants, then the preceeding example becomes:

```
EBUF  COUNT  TYPE  A1!  OK
```

You may have guessed that COUNT is defined in the FORTH dictionary as:

```
:  COUNT  DUP  1+  SWAP  C@  ;
```

To do the equivalent of COUNT TYPE in BASIC:
(assuming string & byte count start at 8000 decimal)

```
100 IF PEEK(8000) = 0 THEN 140
110 FOR X=8001 TO 8001+PEEK(8000)-1
120 PRINTCHR$(PEEK(X));
130 NEXTX
140 REM *** REST OF PROGRAM ***
```

- NUMERIC FORMATTING

Values on the stack may be easily printed. . (dot), U. , and D. have already been mentioned. These three print the number, followed by a space. Since you don't always know how many spaces the number will take up on the screen, column formatting might seem difficut. Fortunately, FORTH provides words for numeric formatting.

.R is used to print a stack value right-justified within a specified field. The number on top of the stack just as .R is executed is used as the field width. The second value on the stack is the value printed out as a SIGNED INTEGER. For instance, suppose we wish to print the top three stack values in a column, each proceeded by a message:

```
:  COLTST   CR   ." VAL1:"  5   .R   CR   ." VAL2:"  5   .R
            CR   ." VAL3:"  5   .R   ;
```

Push three values onto the stack, then execute COLTST. Each number will be printed after the message within a field 5 characters wide, right-justified:

```
21367  17  832  COLTST
VAL1:  832
VAL2:   17
VAL3:21367  OK
```

To print out a number right-justified in BASIC:

```
100 B$=STR$(S): REM **  S = NUMBER TO PRINT
110 PRINT RIGHT$("     " + RIGHT$(B$, LEN(B$)-1 ), 5)
```

U.R is used to print an UNSIGNED number right-justified, and D.R is used to print a SIGNED DOUBLE number right-justified in the same manner.

- SINGLE KEY INPUT

KEY is used to fetch the next character from a line of input. The Commodore 64 input routine is called. If the keyboard buffer is empty, the 64 waits with the flashing cursor input prompt until the RETURN key is pressed. The ASCII values of the keys in the order they were pressed are returned to the KEY routine one at a time. The value for the RETURN key (13 decimal) will indicate the end of the input line. With the way the 64 implements the input function, this routine is not of much use. The following words described will be of more help.

GET calls a C64 routine that scans the keyboard looking for a key depressed. It returns immediately, whether a key was pressed or not. The ASCII value of the key sensed is returned as the top value of the stack. If no key was depressed, then GET returns with a value of zero (0) as the top stack item. Examples of some of these input words will be given in the section LOOPING AND BRANCHING. The BASIC equivalent of GET is:

```
100 GETA$
110 IF A$=""THEN A=0 : GOTO 130
120 A= ASC(A$)
130 REM ** REST OF PROGRAM ***
```

KEYIN, like GET, is not a FORTH standard word. It is provided for convenience in this implementation. A common practice in programs is to print out a message on the screen for the user to read, and wait for a key to be pressed to continue. The trouble with doing this in BASIC is that the flashing cursor does not appear while the program is waiting for a key to be pressed:

```
100 PRINT"PRESS ANY KEY TO CONTINUE"
110 GETA$: IFA$="" THEN 110
120 REM ** REST OF PROGRAM ***
```

or:

```
100 PRINT"DO YOU WISH TO PLAY GAME AGAIN (Y/N) ?";
110 GETA$: IFA$="" THEN 110
120 IFA$="N" THEN STOP
130 IFA$<>"Y" THEN 110
140 REM ** REST OF PROGRAM ***
```

KEYIN provides the same function as line 110 of the above routines, however it also turns on the flashing cursor while it's waiting as an indication to the user.

```
: WAITKEY  CR  ." PRESS ANY KEY TO CONTINUE" KEYIN  ;
```

KEYIN turns off the cursor as soon as a key is pressesd, and returns the ASCII value of the key pressed on the stack. This value may be tested, and program branching may be based on it.

- INPUTTING A LINE OF TEXT

While KEY may be used to input a line of text from the keyboard, there is a cleaner way to do it. But before describing the commands, the description of HERE would be helpful.

HERE is a word that returns the current top of the dictionary, i.e. the address of the first free byte after the end of the dictionary. PAD returns an address that is always 68 bytes beyond the HERE address. When anything is compiled into the dictionary, thus extending it, both HERE and PAD values move upwards. The opposite happens when the size of the dictionary is decreased via FORGET.

The space from PAD downwards to HERE is used for numeric to ASCII conversion, such as in printing a stack value. The space from HERE upwards to PAD is used by the FORTH system for input stream parsing. In other words, the space starting from HERE is used as a temporary character string buffer. Each command in the input stream (that is seperated by spaces) is moved first into the buffer starting at HERE. The character string is assumed to be the name of a FORTH definition. The dictionary is searched for a match. If a match is found, the definition whose name matches the command is executed. If no match is found, the string of characters is converted into a number, if possible, and pushed onto the stack. If conversion is unsuccessful, an error message (? CAN'T FIND) is issued.

The same words that FORTH uses to scan an input line are available to the user. The buffer area starting at HERE is also available, ONLY BETWEEN COMMANDS IN THE INPUT STREAM.

The FORTH word EXPECT may be used to input a string of characters from the user, much like the BASIC:

```
100 INPUT"ENTER MONTH:";A$
```

Before executing EXPECT, the user must indicate where the string of characters is to be placed in memory. HERE is a good place, as long as you make use of all the characters input before exiting back to the FORTH input stream interpreter. Another thing that will corrupt data in the HERE buffer is doing printing of stack values, since numeric conversion starts at PAD and works downwards. This is no problem if the input string is say 40 characters or less, but is a definite problem on longer input lines. The user must also indicate to EXPECT the MAXIMUM number of characters to accept. EXPECT waits until the maximum number of characters is received, or until the RETURN key is pressed. It then puts a couple of <nulls> (0 bytes) at the end of the input string before returning. This allows the user to determine the end of the string.

```
:  ASKMONTH   CR  ." ENTER MONTH:"  HERE  9  EXPECT
              CR  ." THE MONTH ENTERED WAS:"  HERE  9  TYPE  ;
```

The character string may of course be input elsewhere in the FORTH system or moved from HERE for more permanent storage (see the CMOVE command in the glossary).

While EXPECT has it's usefulness, there is a better word. Most of the time you will need a QUALIFIER to a word. For instance, suppose you wanted to print a page header on the printer after opening it as a file. In BASIC you would do:

```
OPEN4,4,1
PRINT#4,"LISTING OF STAR SPARKYS SPACE WAR GAME"
```

FORTH has a word called WORD that will scan off a string of characters from the input stream and place them into HERE, preceeded by a byte count. The DELIMITER CHARACTER may be specified. This means that you specify the character that the text string may start and end with. This may be the space character, the quote character, or anything else.

When the word definition using WORD is executed, it must be followed by a text string that starts and ends with the specified delimiter. The file I/O commands are explained in another section, but to just explain the next example, the PRINT# routine requires the file#, address of a text string, and a text string byte count (file# addr count ---):

```
( define command to output following string to printer )

: PRNTR  34  WORD        ( SCAN OFF STRING WITHIN QUOTE CHARACTERS)
   4                     ( FILE# )
   HERE                  ( START OF STRING LEFT BY WORD )
   COUNT                 ( LEAVE ADDRESS & LENGTH OF STRING )
   PRINT#  ;             ( SEND STRING TO PRINTER )
```

Once defined, PRNTR may be used to send a character string to the printer:

```
4 4 0 0 0 OPEN
PRNTR  "LISTING OF STAR SPARKYS SPACE WAR GAME"
```

(if you tried this with a COMMODORE VIC 1525 printer, you'll have to output a RETURN to get the line to print. Type the following sequence:)

```
13  EBUF  C!  4  EBUF  1  PRINT#
4 CLOSE
```

- NUMERIC INPUT

While it is easy to push numeric values onto the stack before executing a FORTH word, it is often desirable to have a routine request a numeric value from the user. Since the user is typing alphanumeric characters, how can these be converted into numbers? In BASIC this is done automatically:

```
INPUT"ENTER STARTING NUMBER:";N
```

In FORTH, NUMBER converts a string of ASCII characters in memory into a SIGNED DOUBLE NUMBER which it leaves on the stack. The address of the FORTH character string (byte count + string) is passed to NUMBER. The number is converted according to the current numeric BASE. If a decimal point is encountered in the character string, its position is stored in the user variable DPL, but will otherwise be ignored. Since the conversion stops upon reaching a space or a null byte, EXPECT may be used to input the string as long as a byte count greater than or equal to the length of the string is placed as the first byte. An ASCII minus sign encountered at the start will leave a negative value. A general numeric input routine may now defined:

```
(  ---  d  )
: INPUT   DECIMAL                          ( or any other base )
          HERE  1+  10  EXPECT     ( read in up to 10 characters )
          10  HERE  C!               ( place byte count at start )
          HERE  NUMBER  ;             ( convert string to number )
```

To input a single precision integer, just DROP the top word off the stack after INPUT returns, or place DROP in INPUT just after NUMBER and before ; .

```
INPUT  1234  OK
. 0  OK
. 1234  OK
```

Since WORD may also be used to input numeric values, it is easy to read in and convert numeric strings representing the values in a table. For instance, in the graphics command set there is a word called SPDEFINE. SPDEFINE is used to create a sprite definition in memory. SPDEFINE calls the definition SPCODE to read in and convert 63 numbers (actually character strings representing numbers) that follow in the remainder of the screen. SPCODE is defined in the dictionary as:

```
:  SPCODE
   21  0  DO                                    ( DO FOR 21 LINES )
     3  0  DO                            ( DO FOR 3 VALUES PER LINE )
       BL  WORD  NUMBER  DROP            ( READ IN & CONVERT NUMBER )
     LOOP
   SWAP  ROT        ( REVERSE ORDER OF THE 3 BYTES IN THIS LINE )
   C, C, C,             ( COMPILE THE BYTES INTO THE DICTIONARY )
   LOOP  ;
```

If a screen contains the following:

```
HEX   CREATE  SPRO  SPCODE
      00 00 00
      00 FF 00
      01 FF 80
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      03 00 C0
      01 FF 80
      00 FF 00
      00 00 00
```

When this screen is loaded, a new dictionary entry called SPRO will be created and SPCODE will read in the following 63 values and store them as bytes in the body of the definition. By modifying the above definition of SPCODE you can create varying sizes of data tables, arrays, etc.

HANDLING STRINGS IN FORTH

The techniques previously mentioned for text input and output are available in most FORTHs. C64-FORTH provides a special group of words to manipulate character strings for the convenience of the user. They are not normally part of the C64-FORTH system - they must be loaded in from disk. On your C64-FORTH disk starting on screen # 20 are the source definitions for the string words. Just type:

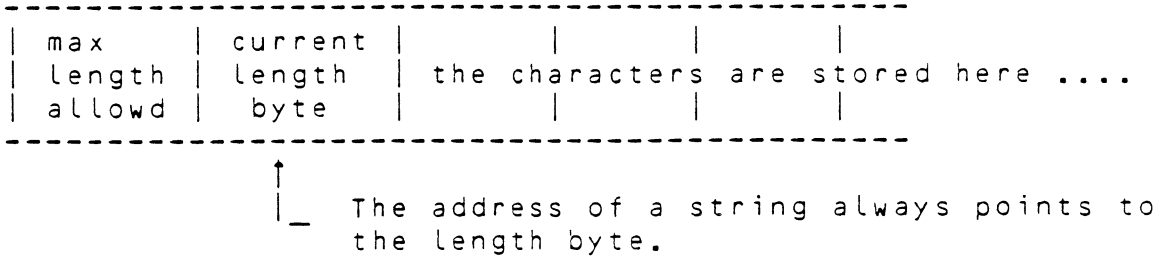
```
20 LOAD
```

and the extensions will be added onto your current FORTH system. The commands described in the rest of this chapter are now available for your use.

- HOW STRINGS ARE STORED IN MEMORY

It was previously mentioned that FORTH strings are stored as a string of characters preceeded by a length byte (note that EXPECT and TYPE are exceptions in that they deal only characters themselves, and it is up to the user to keep track of byte counts). The string extensions follow a similiar concept, however the current-length byte is preceeded by a maximum allowable length byte. The maximum allowable length for ANY string is 255 characters. Most of the string functions return an address which points to the current-length byte. Normal FORTH words such as COUNT, TYPE, CMOVE, etc. can be used to operate on strings in memory. However, when a string is stored into memory, the byte preceeding the destination is checked. If the length of the string to be stored is longer than the maximum count allowed, then only the allowed number of bytes is accepted from the string. This feature prevents the user from overwriting parts of the FORTH system by moving strings too long for the destination - a precaution most FORTHS do not provide.

A FORTH string in memory:



The space starting at PAD is used as a temporary storage area for strings. If a string is currently stored starting at PAD, PAD will contain the current length of the string, the string itself will start at PAD +1 and continue upwards, and NO max byte count is stored at PAD -1. PAD is always assumed to be able to contain the maximum length for a string - 255 characters. Certain string functions also use the space from PAD+256 to PAD+511 for temporary operations, but this is transparent to the user.

The C64-FORTH extensions manipulate strings in a manner similiar to numeric stack items. If the stack has a limited depth, how can it be used to hold strings also? Easy - the strings themselves are not stored on the stack, rather the ADDRESS of the string is stored on the stack. Normal stack operators such as DUP, ROT, SWAP, etc. may be used to manipulate string pointers. This allows for very efficient string handling.

A few things must be kept in mind - One of the functions, $!, checks the specified destination address, and uses the preceeding byte as a max byte count. Therefore, you cannot use $! to move a string to some random memory location that has not been declared as a string variable or constant UNLESS you first store a max byte count into the destination address-1. You could however use CMOVE or <CMOVE to move the string there. Then any of the string functions can use that string as a SOURCE string for an operation, since the max length count is only checked when storing a string. Also, a null string is indicated ONLY by the current length byte being equal to 0 (zero). The contents of a null string may be any random pattern of byte values so only the current length = 0 should be used to detect a null string. One thing more - the address of PAD changes with the current end of the FORTH dictionary. If a string is temporarily stored at PAD, and the end of the dictionary is changed, the data starting at the new location of PAD is meaningless.

In the following descriptions, the same stack notation is used to show the stack requirements and effects as is used in the glossary. $1, $2, ... will represent strings (actually the address of the strings) on the stack.

```
-  "                    (       ---            at compile time
                                ---  $1         at run time       )
```

The first of the string extentions is " (quote). " functions similiar to ." except instead of printing a string, " moves a string into the temporary holding area at PAD. Like ." , " operates differently in direct (interpretive) mode than when being compiled into a definition. To put a string of characters into the PAD area, just type " followed by one space, then the string of characters delimited by another " character:

```
" ABCDEFG" OK
```

" leaves the address of PAD on the stack once it stores the string there. To see the character string as it is stored at PAD type:

```
8 DUMP          ( the address of PAD was left on the stack )
 5206   7  41  42  43  44  45  46  47   .ABCDEFG
```

(DUMP displays a section of memory starting at a specified address for a specified # of bytes. Regardless of what base you are in, DUMP always prints in HEX numbers to give a readable format on the 40-character screen. The contents of 8 locations are printed per row, followed by ASCII representations of the contents of the locations.)

" can also be used within a colon definition. When the definition is being compiled, the string is placed into the definition. When the definition is executed, the ADDRESS of the string in memory (in the body of the definition) is placed on the stack:

```
: TEST  "  TEST STRING"  ;  OK
```

Now type:

```
TEST  OK
U.  24872  OK
```

The address printed will be dependent upon the actual location in memory the definition TEST ended up. You may print the string by:

```
TEST  COUNT  TYPE   TEST STRING  OK
```

```
-  $.                    (  $1  ---     )
```

The COUNT-TYPE pair seems inconvenient. There are words for printing out stack values, so one is provided for printing out strings from a string address on the stack. $. prints the string pointed to by the top address on the stack. The address of the string is removed from the stack by $. From a previous example:

```
TEST  $.  TEST STRING  OK
```

- $VARIABLE (N --- at compile time
--- $1 at run time)

$VARIABLE is used to create a new dictionary entry, and reserve space for a string of a specified maximum length. When creating a new string variable, you specify the maximum length of any strings to be stored in it in the future, plus the name of the variable. For example:

16 $VARIABLE FILENAME OK

will create a string variable that can hold a string of up to 16 characters in length. The contents of the variable just created is not initialized, and may consist of random values, but the current string length is set to 0 (zero) to indicate a null string is currently stored. At any future time, just referencing the name of the string variable will push the address of the string contained onto the data stack. Examples of using string variables are given in some of the other word descriptions.

- $CONSTANT ($1 --- at compile time
--- $2 at run time)

$CONSTANT functions just like $VARIABLE except it provides a way to pre-initialize a string variable. Once the string constant is created, it appears exactly like and can be used like, a string variable. The maximum length of the string constant defined is set to the length of the string it was initialized to. To create a string constant, first put the address of a string on the data stack, then use $CONSTANT followed by the name you wish to assign to the new constant:

" Press any key to continue..." $CONSTANT PKEY OK

" defined a string and left the address of PAD on the stack, which is where the string was temporarily stored. $CONSTANT then created a new dictionary entry, called it PKEY, and moved the string at PAD to PKEY. To verify this:

PKEY $. Press any key to continue... OK

Even though it is called a constant, you can store a string into PKEY at a future time, since it acts like a string variable once it is created.

- $! ($1 $2 ---)

$! stores string $1 at the location $2. If $2 is the address of PAD, then the maximum byte count of the destination is assumed to be 255; otherwise the byte at $2-1 is fetched and used as the max count. The MINIMUM of either the current length of $1 or the max byte count just fetched is the number of characters transferred to the destination. The length of the string formed at $2 is set to the number of characters actually moved.

From a previous example:

```
" TESTDATA.DAT" FILENAME $! OK
FILENAME $. TESTDATA.DAT OK
```

Let's try storing a string that is longer than the maximum length allowed. When creating the string variable FILENAME in a previous example, we specified a maximum length of 16 bytes or characters.

```
" ABCDEFGHIJKLMNOPQRSTUVWXYZ" FILENAME $! OK
FILENAME $. ABCDEFGHIJKLMNOP OK
```

Here it is obvious that only 16 characters were accepted when storing the specified string into a string variable of only 16 characters maximum length.

- $+ ($1 $2 --- PAD)

$+ is used to concatenate (add together) strings. $2 is added onto the end of $1 and the resulting string is left starting at PAD. Either string may be in PAD to begin with (but not both obviously), and the resulting string is left in PAD.

Examples:

```
" ABCDEF" $CONSTANT TEMP$ OK
" 123456" TEMP$ $+ OK
$. 123456ABCDEF OK

TEMP$ " 123456" $+ $. ABCDEF123456 OK
```

If the total length of the two strings combined is greater than 255 characters, an erroneous string will result.

- LEFT$ ($1 N --- PAD)

LEFT$ functions similiar to its BASIC companion. LEFT$ returns a string comprised of the left N # of characters of string $1. If the length of string $1 is less than N characters long, then the string $1 in its entirety is returned. The string $1 may be at PAD to start with, and the resulting string is left at PAD.

```
" 1234567890" 4 LEFT$ $.  1234  OK
TEMP$ 8 LEFT$ $.  ABCDEF  OK
```

Remember that when TEMP$ was defined a few examples ago, it was initialized with a string of only 6 characters, therefore the full string was returned when the leftmost 8 characters was requested.

- RIGHT$ ($1 N --- PAD)

RIGHT$ works similiar to its BASIC counterpart. The rightmost N # of characters is returned from string $1 with the resulting string left at PAD. If N is = 0, then a null string is returned, if N is greater than the length of $1, then $1 is returned.

```
TEMP$ 3 RIGHT$ $.  DEF  OK
TEMP$ 12 RIGHT$ $.  ABCDEF  OK
```

- MID$ ($1 STRTPOS #CHR --- PAD)

MID$ returns a sub-string comprised of #CHR characters starting at position STRTPOS in string $1. As in BASIC, MID$ assumes the first character in a string is at position 1, the second character is in position 2, etc... If STRTPOS is greater than the length of $1, or if either STRTPOS or #CHR is equal to 0, then a null string is returned. If string $1 has fewer characters than #CHR from the starting position to the end of the string argument, then the whole rest of the string is returned.

```
" 1234567890" 3 4 MID$ $.    3456  OK
" 1234567890" 10 1 MID$ $.     0  OK
" 1234567890" 12 3 MID$ $.    OK     (null string was
                                      returned)
```

- LEN ($1 --- LEN)

LEN returns the current length of the specified string. To use a previous example:

```
TEMP$ LEN . 6 OK
" 1234567890" 5 0 MID$ LEN . 0 OK   (a null string was
                                      returned by MID$)
```

- CHR$ (B --- PAD)

CHR$ returns a 1-character string created from the byte value passed on the stack. This may be used to insert carriage-returns in a string, printer control commands, etc.

```
TEMP$ 13 CHR$ $+ $. ABCDEF
 OK
```

- ASC ($1 --- B)

ASC returns a number from 0-255 which corresponds to the Commodore ASCII value of the first character in the specified string. If there are no characters in the string, a "NULL STRING" message will be printed and a value of zero will be returned. The null-string message can be avoided by adding (concatenating) a string equal to CHR$(0) onto the end of the string to use ASC on.

```
" ABCDEF" ASC . 65 OK
" " ASC . NULL STRING 0 OK
0 CHR$ TEMP$ $! " " TEMP$ $+ ASC . 0 OK
```

- VAL ($1 --- N)

VAL converts a string of characters into a single-precision number and returns it on the stack. The conversion stops at the first non-digit character. The character string may start with a minus sign, and the converson is done according to the current base.

```
" -1234" VAL . -1234 OK
" 65530" TEMP$ $! TEMP$ VAL . -6 OK
```

- DVAL ($1 --- D)

DVAL works just like VAL except a double-precision # is returned:

```
" 12345678" TEMP$ $! OK
TEMP$ DVAL D. 12345678 OK

HEX TEMP$ DVAL DECIMAL D. 305419896 OK
```

Note that in the last example the conversion pushed a value equal to 12345678 HEX onto the stack, which when printed in decimal comes out 305419896.

- STR$ (N --- PAD)

STR$ takes the top stack value and converts it to a string of characters, leaving it at PAD. The resulting string begins with either a space or minus sign depending on the sign of the original value.

```
1234 STR$ 8 DUMP
  6592   5  20  31  32  33  34  20  20   . 1234     OK

-32760 STR$ 8 DUMP
  6592   6  2D  33  32  37  36  30  20   .-32760    OK
```

- DSTR$ (D --- PAD)

DSTR$ acts the same as STR$ except a double-precision number is taken from the stack and converted to a string. No decimal point is included in the string, however a definition can be created for that purpose :

```
12 $VARIABLE S1  12 $VARIABLE S2   ( temporary variables )
: DLRSTR$                          ( D --- PAD )
  DSTR$ " $" S1 $+ S1 $!
  S1 DUP LEN DPL @ - LEFT$ S2 $!
  S1 DPL @ RIGHT$ S1 $!
  S2 " ." $+ S1 $+ ;               ( string left at PAD )
```

DLRSTR$ may now be used to create a dollar value out of a double number on the stack (note: the position of the decimal point must be in variable DPL, which it already is if the double number on the stack has been the most recent number converted from the input stream) :

```
534.86  DLRSTR$  $.  $ 534.86  OK
```

The DPL @ pair used in DLRSTR$ may be replaced by the constant 2 to ignore the position of the decimal point in the number input to always have 2 decimal places in the final string. S1 2 255 MID$ S1 $! may be used after DSTR$ in DLRSTR$ to eliminate the leading space thus putting the dollar sign right before the value, and other improvements may be made in a similiar manner.

- INPUT$ ($1 ---)

INPUT$ provides very convenient way to input a character string and store it in a variable. In BASIC we would do :

```
10 INPUT A$
```

INPUT$ allows us to do a similiar thing in FORTH. The FORTH word EXPECT is called, which requires a maximum count of characters. If the address to store the string being input ($1) is equal to PAD, then 255 is used as the input count; otherwise the maximum length of the destination is used.

```
80  $VARIABLE  A$
:  GETDATE  CR  ." INPUT TODAY'S DATE: "  A$  INPUT$  ;
```

When GETDATE is executed, a character string of up to 80 characters (the maximum length of string variable A$) will be accepted and stored in A$.

```
GETDATE
INPUT TODAY'S DATE:  JULY 27, 1983   OK
A$  $.  JULY 27, 1983  OK
```

- POS$ ($1 $2 --- N)

POS$ searches for the first occurance of string $2 within string $1. If found, N is returned which is the position of the start of the matching string within $1. If not found, N returns equal to zero. Using the string input into A$ in the previous example:

```
A$  " 27,"  POS$  .  6  OK
A$  " 1983A"  POS$  .  0  OK      ( no match found )
A$  " JULY"  POS$  .  1  OK
```

- $= ($1 $2 --- flag)

$= is used to compare two strings and returns a true flag if the strings contain the same characters AND they are equal in length.

```
" ABCDEFGH" A$ $!  OK
" ABCDEF" A$ $= .  0  OK   ( strings not same length )
" ABCDEFGH" A$ $= .  1  OK    ( strings identical   )
```

- $< ($1 $2 --- flag)

$< compares $1 to $2 and returns a true flag if $1 is less than $2, OR if $1 is shorter than $2.

```
" ABCDEF" A$ $< .  1  OK   ( $1 is shorter than $2 )
" ABCDEFGH" A$ $< .  0  OK  ( $1 is not less than $2 )
" ABCDEFGG" A$ $< .  1  OK  ( $1 is less than $2 by values)
```

- $> ($1 $2 --- flag)

$> compares $1 to $2 and returns a true flag if $1 is greater than $2, OR if $1 is longer in length.

- $NUMBER ($1 --- D)

$NUMBER functions almost identical to the FORTH standard word NUMBER. However, NUMBER aborts with an error if the string being converted into a double number contains an unexpected character. $NUMBER converts a string of characters into a double-precision stack value, but only stops the conversion process upon encountering a non-expected character in the string. 'Expected' characters means a possible leading minus sign, an embedded decimal point, and any numeric value legitimate to the current base. The address of the first non-expected character within string $1 that the conversion stops at is left in the variable $NUMADR.

$NUMBER will prove very useful at times. If you have created a data file from within a BASIC program, and PRINTed numeric values out to it, you will have a data file containing something like this on disk:

```
 34, 567,JOHN SMITH,1428 ELM ST.,LIVELYHOOD,KA,999-555-8426
```

The above may have been created with a BASIC statement such as:

```
2020 PRINT N;B;N$;T$;C$;S$;P$
```

BASIC seperates PRINTed values by commas, which are not 'expected' characters in NUMBER conversion, but $NUMBER will convert one of the numbers in the above example and stop the conversion process upon encountering a comma. The address of the character that stopped the conversion, in this case one of the commas, is left in $NUMADR. A combination of the value left in $NUMADR, and proper use of MID$ can be used to scan off and convert a data file saved on disk via either BASIC or C64-FORTH.

OTHER DEFINITIONS:

Following are a few other definitions used in the string extension screens that may occasionally be useful:

- -TEXT (addr1 N1 addr2 --- N2)

-TEXT is a PROPOSED FORTH STD word. The description of the proposal (and the operation of the word as defined in the C64-FORTH string extensions) is: "Compare two strings over the length N1 beginning at addr1 and addr2. Return zero if the strings are equal. If unequal, return N2, the difference between the last characters compared: addr1(i) - addr2(i) ".

Note that the addresses passed to -TEXT are NOT normal string address (i.e. of the byte count at the start of a string), but rather absolute addresses within strings of characters.

- $COMPARE ($1 $2 --- flag)

Compares $1 with $2 returning a flag as follows:

1) Positive if $1 is greater than $2 OR equal to $2 but longer,
2) Zero if the strings are identical in content and length,
3) Negative if $1 is less than $2 OR equal to $2 but shorter.

- UMOVE (fromadr toadr #byt ---)

UMOVE is a universal version of CMOVE and <CMOVE. UMOVE decides which version to call, taking into consideration source-destination overlap.

FILE I/O

The FORTH screens generated by C64-FORTH are saved onto disk or tape as sequential data files. The routines used by C64-FORTH to do this are all available to the user. They allow creating or accessing FORTH screens as data files, and data files created by or to be used by BASIC programs. As in BASIC, tape I/O is as easily worked with as disk I/O. Program type files and relative files also are accessable.

The one major difference between accessing data files via C64-FORTH and through BASIC is that C64-FORTH does data reading and writing from/to an external file on a straight byte-by-byte basis. It is up to the user to determine and identify whether data is numeric, components of ASCII string data, etc. Conversely, error checking is easier than possible in BASIC.

- SYSDEV#

SYSDEV# is a C64-FORTH variable used by the I/O routines. Most routines automatically set it to the device number of the I/O unit currently being accessed. The low-level routines of OPEN, CLOSE, GET#, INPUT#, PRINT#, AND CMD# do not use SYSDEV# at all, they rely upon the device number being passed through the OPEN command. However, some of the disk error checking routines do require that the device number of the disk be present in SYSDEV#. It is a good practice to set SYSDEV# before opening any files to a disk or tape device. Note that the low byte of the contents of SYSDEV# hold the device #, and the high byte, normally 0, indicates which drive to use of a multiple-drive unit such as a 4040. Also, SAVESYSTEM and SAVETURNKEY commands require that SYSDEV# be holding the device number of the I/O unit that C64-FORTH is to be saved on.

- ST

ST is a C64-FORTH variable. After any I/O using OPEN, CLOSE, GET#, INPUT#, and PRINT#, the ending I/O status is left in ST. In BASIC programs, the variable ST is usually checked after a transfer:

```
100 INPUT#2, A$(I),B          : REM ** READ IN STRING & NUMBER
110 IF ST<>0 THEN 1000        ; REM ** LINE 100 IS ERROR ROUTINE
```

The same can be done in C64-FORTH:

```
... 2 HERE 80 INPUT# ST C@ IF ." I/O STATUS ERROR" ...
```

- OPEN (file# dev# sa fnptr fnbytcnt ---)

A file must be opened before it may be accessed. Parameters must be passed to the OPEN command giving the specifications necessary to access the file. The major difference between the FORTH OPEN and the BASIC OPEN commands is that with BASIC OPEN, an optional filename or command string may be specified, whereas the FORTH OPEN must be passed a filename/command string count and pointer EVEN IF ONE DOES NOT EXIST (in this case a 0 is pushed onto the stack for a filename/command string count, and any value may be pushed on as a filename/command string pointer, as this will be ignored).

(assuming a filename/command string and byte count is in a data type called FNBUF and FNBUF contains the byte count of 14 followed by the string '0:TESTDATA,S,R')

```
2                ( push a file# onto stack )
8                ( push device # onto stack )
2                ( push secondary address onto stack )
FNBUF  COUNT     ( push filename string address and byte count )
OPEN             ( open the device for input )
```

BASIC equiv:
OPEN 2,8,2,"0:TESTDATA,S,R"

(to open a device without specifying a filename or command string)

```
4                ( push file# onto stack )
4                ( push device # )
1                ( push secondary address )
0                ( push any value to keep stack balanced )
DUP              ( push filename byte count - 0 in this case )
OPEN             ( open the device )
```

BASIC equiv:
OPEN 4,4,1

- CLOSE (file# ---)

Any file that was opened must be properly closed to ensure all data is written to the file, and to free up the file or device for further use.

```
2  CLOSE         ( close file # 2 )
```

BASIC:
CLOSE 2

- INPUT# (file# addr count ---)

This is the easiest way to retreive character data from a file on disk or tape. The count specifies a MAXIMUM number of characters to input (BASIC automatically limits at 80). INPUT# terminates before reaching the maximum count if a RETURN (byte value =13), a <null> code (byte value =0), or the END-OF-FILE status is received. The data received, including the RETURN or NULL byte if encountered, is placed in memory starting at <addr+1>. The count of the actual number of bytes received (this does not include the RETURN or NULL bytes) is placed at <addr>, forming a normal FORTH string.

```
2  HERE  80  INPUT#    ( read 80 characters max. into memory
                         starting at end of dictionary )
```

```
BASIC:
INPUT#2,A$ or INPUT#2,C
```

- GET# (file# addr count ---)

Similar to INPUT#, GET# retrieves <count> number of bytes from <file#> and stores them as a FORTH string (byte count + bytes received) starting at <addr>. Input is terminated before reaching maximum count ONLY by detection of the END-OF-FILE status. GET# can be used instead of INPUT# when the input data is expected to contain byte values equal to 0 or 13 decimal. GET# is more useful than BASIC's GET# because it can be used to input more than one byte at a time. Since the setup for a transfer of a byte is far greater than actually transferring the byte, multiple byte GETs will be very fast, compared to putting GET# in a loop in BASIC.

```
( fetch 20 bytes from data file )

7  HERE  20  GET#
```

```
In BASIC:
100 FOR Y=1 TO 20
110 GET#2,A$ : A$=A$+CHR$(0)
120 A = ASC(A$)
130 POKE AD,A
140 NEXTY
```

- PRINT# (file# addr count ---)

PRINT# is used to transfer data bytes to an external file. <count> number of bytes are transferred starting from <addr>. If a FORTH string starts at HERE, then to send it to file# 5:

```
5  HERE  COUNT  PRINT#
```

This is similar to BASIC's: PRINT#5,A$

NOTE: 5 0 0 PRINT# is NOT identical to BASIC's PRINT#5 by itself, since Commodore 64 BASIC automatically outputs a RETURN after a print statement that doesn't end with a semi-colon. To do the identical function, place a RETURN code somewhere in memory, and output that:

```
13  EBUF  C!  5  EBUF  1  PRINT#
```

- CMD# (file# ---)

CMD# is similar to BASIC's CMD with a slight difference (hence the different name); output is not diverted from the screen to the external device, but is sent out seperately after it is printed on the screen. In other words, the external device is opened, a byte transmitted, and then closed each time a byte is printed on the C64 display. Since disk versions of FORTH rely heavily on the disk, this was necessary. Otherwise, to list multiple screens for instance, a screen would have to be read in, the CMD file opened, CMD issued, the screen printed, the CMD device closed, the next screen read in from disk, etc. Disk directories would have to be buffered in memory. CMD# implements this in a much cleaner way allowing true hardcopy of the screen display.

For an example of usage, see the section under 'Details of the C64-FORTH system' called 'Listing multiple screens on the VIC-1525 printer'.

- ABORTIO

This routine may be called from any FORTH routine. It closes ALL open files. This may be used in case of an I/O error.

- GETDEV (file# --- dev#)

GETDEV returns the device number that a specified file number was opened with. The file number must be a currently open file.

- EBUF (--- adr)

EBUF is a 30-byte long buffer used to hold the disk status sent from the disk. See RDSTAT, CKRDSTAT, and CKWRSTAT.

- RDSTAT (file# ---)

RDSTAT reads in the disk status from the disk ONLY if file# refers to a physical device number > 3. It is up to the user NOT to use this routine on any device that is not a disk or cassette. The status read in from the disk is left in EBUF as a FORTH string. If the device is a cassette (i.e. the device # < 4), then an ASCII "00" (zero-zero) is left at EBUF+1 and +2. Since the first two ASCII bytes returned as a disk status indicates the error code, a FORTH routine may be generalized with regard to disk or cassette. RDSTATting from a cassette will return an 'OK' status (indicated by the two ASCII zeros).

- CKST (--- flag flag=0 if ok, =1 if end-of-file)

CKST can be used to check the I/O status after a transfer (see ST). If the status byte was equal to 0, a FALSE flag is returned. If an END-OF-FILE is detected, this is not usually an error but an indication for the program, hence CKST will return TRUE. Any error in the status, and CKST does not return. It prints the error message 'I/O STATUS ERROR: ' and prints the status byte in the current BASE. CKST then returns execution to the FORTH input interpreter.

- CKRDSTAT (--- flag flag=0 if ok, 1 if end-of-file)

CKRDSTAT first calls CKST to check the I/O status byte. Then it reads the ERROR CHANNEL to the disk (OPENCHN opens the error channel as file# 15). CKRDSTAT returns FALSE if everything is ok, TRUE if end-of-file was detected. If the disk status indicates an error (i.e., an error code >=20 is returned), then the error message received from the disk (and stored in EBUF) is printed. All open I/O files are closed, and the C64-FORTH disk error message is printed (ERROR is executed). Control returns to the FORTH input interpreter.

- CKWRSTAT (---)

CKWRSTAT is identical to CKRDSTAT except no flag is returned. This is not necessary when checking disk status after WRITING to the disk, since the end-of-file indication is meaningless and will never occur. CKWRSTAT can be used to check disk status when reading, if you know the end-of-file status will not occur, or you expect it to occur reading the last byte of a file of known length.

- ?DISK (--- flag flag=0 if not disk, <>0 if disk)

?DISK returns a flag based on whether the device number in SYSDEV# is a disk or not. The only determination made is whether the device number is >= 4.

- OPENCHN (---)

OPENCHN opens the error channel to the disk. OPENCHN is defined as:

```
:  OPENCHN  ?DISK  IF  15 SYSDEV# C@ 15 OPEN  CKWRSTAT  THEN  ;
```

In BASIC, this would be:

```
10 OPEN 15,8,15
20 IF ST<>0 THEN PRINT"I/O STATUS ERROR:";ST : STOP
30 GOSUB 1000
1000 INPUT#15,EN,EM$,ET,ES
1010 IF EN=0 THEN RETURN
1020 PRINT"DISK ERROR ",EN;EM$;ET;ES : STOP
```

- CLOSCHN (---)

CLOSCHN closes the error channel to the disk that was opened by OPENCHN. CLOSCHN is defined as:

```
:  CLOSCHN  ?DISK  IF  15  CLOSE  THEN  ;
```

```
- ?FILE    ( ---  flag      flag=0 if ok, =1 if file not found )
```

?FILE is handy after opening a file to see if it exists on disk, or if the 'file does not exist' code is returned by the disk. ?FILE is defined as:

```
HEX  :  ?FILE  CKST  DROP  ?DISK  IF
          15  RDSTAT  EBUF  1+  @  DUP  3236  =  IF
             DROP  1  ELSE  3030  -  IF
                CR  EBUF  COUNT  TYPE
                CR  ABORTIO  8  ERROR
             THEN
          ELSE  0  THEN  ;
```

```
**********************************
*****   SAMPLE PROGRAMS   *****
**********************************
```

This is a good spot to give some sample programs. All three of the following examples require the C64-FORTH string extensions.

#1 - Sending a command to the disk

The C64-FORTH word DC can be used to send a command to the disk, however, it cannot be compiled into a definition. To define a version that can be compiled into a definition:

```
:  DCMD                                  ( $adr  ---  )
( DCMD sends the string at the address on the stack to the disk )
( Note: do not use this to send the directory command          )
     15  SWAP  COUNT  PRINT#             ( send string )
     CKWRSTAT  ;                         ( check disk status )
```

Now, lets say we happen to have a collection of files on the disk, called 'CLIENT1', 'CLIENT2', 'CLIENT3', 'CLIENT4', and 'CLIENT5'. We had written a simple filing system, which had left these temporary files out on the disk. Each one contains all the data relating to one client account. We have dropped client #1, so we wish to drop his data, and consider client #2 to be the new client 1, etc. The following routine will delete CLIENT1, and rename the subsequent files.

```
20  $VARIABLE  N$                       ( define temporary string var. )
:  REMOVECLIENT1
     OPENCHN                            ( open error channel to disk   )
     " S0:CLIENT1"  DCMD                ( remove client 1 file )
     6  2  DO                           ( set up DO LOOP to rename files )
       " R0:CLIENT" N$ $!               ( form rename command string )
       I STR$ 2 255 MID$ N$ SWAP $+ N$ $!
       N$ " =CLIENT" $+ N$ $! N$ I 1+ STR$ 2 255 MID$ $+
       DCMD                             ( send command string to disk )
     LOOP
     CLOSCHN   ;                        ( close error channel )
```

#2 - Print the starting and ending load addresses of a PRG file

The following routine requests the name of a file on the disk in dirve #8, then prints out the starting and ending load addresses of that file.

```
16 $VARIABLE N$                    ( temporary string variable )
: LOADADR
  CR ." INPUT FILENAME: "          ( get name of file from user )
  N$ INPUT$
  CR ." LOAD ADDRESSES OF " N$ $. ." ARE..." CR
  OPENCHN                          ( open error channel )
  14 8 14 N$ " ,P,R" $+ COUNT OPEN      ( open the file )
  ?FILE IF                         ( does the file exist ? )
    CR ." FILE DOES NOT EXIST"           ( no )
  ELSE
    HEX 14 HERE 2 GET#             ( yes, get starting load address )
    CR ." STARTING ADR: "          ( and print it )
    HERE 1+ @ DUP 4 U.R CR         ( leave address on stack )
    BEGIN                          ( read in all bytes in the file )
      14 HERE 1 GET#               ( one at a time and increment the )
      1+                           ( address for each byte )
    CKST  UNTIL                    ( end-of-file ? )
    1-                             ( yes, back address up one )
    ." LAST BYTE ADDR: " 4 U.R              ( and print it )
  THEN
  CR 14 CLOSE CLOSCHN DECIMAL ;          ( close channels )
```

#3 - Copy a screen the hard way - a 1024 byte file-to-file copy.

The following routine copies the contents of one C64-FORTH screen to another. While the C64-FORTH command COPY will do this nice and easy, this is a simple example of data I/O to and from the disk.

```
3 $VARIABLE A$  3 $VARIABLE B$  10 $VARIABLE C$
: CKFILE            ( ---  flag    flag =0 if file wasn't found)
  ?FILE IF
    CR ." FILE CALLED " PAD $. ." NOT FOUND" 0
  ELSE  1  THEN  ;

: COPYSCR                              (  ---  )
  CR ." SCREEN # TO COPY FROM: "  A$  INPUT$
  CR ." SCREEN # TO COPY TO  : "  B$  INPUT$
  OPENCHN
  " SCR" A$ $+ C$ $!
  14 8 14 C$ " ,S,R" $+ COUNT OPEN       ( open input file )
  CKFILE IF                              ( does file exist ? )
    " SCR" B$ $+ C$ $!
    13 8 13 C$ " ,S,W" $+ COUNT OPEN     ( yes, open output file )
    CKWRSTAT
    14 HERE    1024 GET#    CKWRSTAT     ( read in 1K of data )
    13 HERE 1+ 1024 PRINT#  CKWRSTAT     ( write it out to file )
  THEN
  13 CLOSE 14 CLOSE CLOSCHN CR ;         ( close all files )
```

- SAVESYSTEM and DC

Every FORTH user develops his or her own custom word definitions to do common functions easily. The desire to make them a permanent part of the FORTH system soon arises. Instead of booting up the original version of C64-FORTH each session and loading in the screens with the definitions, it is very easy to save the current system with all new additions onto disk. The SAVESYSTEM command is used as follows:

```
SAVESYSTEM  "MYFORTH"
```

SAVESYSTEM first prompts the user to insert the ORIGINAL C64-FORTH disk into the drive. SAVESYSTEM reads a section of it, then prompts the user to insert the disk the FORTH system is to be saved on.

The current version of the FORTH system will be saved onto disk as a file named "MYFORTH" (or whatever filename you put between the quotes). The next time you power-up the system just load "MYFORTH" (or whatever you called it), and RUN. Do a VLIST and you will see that your custom definitions are part of the FORTH system.

The string between the quotes is sent directly to the disk or cassette, so disk file replace functions work:

```
SAVESYSTEM  "@0:FORTH2.EXE"
```

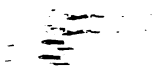
C64-FORTH provides a word to send commands to the disk when in the interpretive mode. DC sends whatever is between the quotes following it to the disk, and reads the status back. DC can be used alone without a following string to just read back and check the status. The status is not printed if it has a status value less than 20. In the current 1541's, the only two status codes returned less than 20 are 'OK' and 'FILES SCRATCHED'. Examples:

```
DC  "N0:FORTHWORK,01"        ( send cmd to format a disk )
DC  "S0:TEMP.DAT"            ( scratch file )
DC                           ( just read and check status )
```

The user should never use the original C64-FORTH disk, except to load C64-FORTH for the first time, and for SAVESYSTEMing and SAVETURNKEYing. By using the DC command along with SAVESYSTEM, you can easily format and save the FORTH system off onto a new disk. A disk with the C64-FORTH program on it appears as any other Commodore-64 program disk. Screens are saved as ASCII sequential data files. This means you can save other kinds of programs as well as data files created by BASIC programs, relative files, etc. By changing just a variable or two, C64-FORTH can be quickly adapted to make full use of an extended capacity floppy or Winchester drive.

Screens may be moved from one disk to another by writing a routine to read them in from one disk (it would help to allocate more disk buffers via the BUFFERS command), mark them as updated, then place the new disk in the drive and execute SAVE-BUFFERS.

There is no initialization of screens necessary for a NEW'ed blank disk. If a screen or block is requested, and the specified screen does not exist on disk, a screen full of blanks is delivered to the user. Only if the blank screen is altered or marked as updated, AND SAVE-BUFFERS is executed, will it be placed on the disk. (See the description of TAPEFLG for how this works with cassette)



- DIR

DIR displays the directory of drive 0 of the disk specified in SYSDEV#. DC "$" also calls the routine DIR. DC "$1" may be used for the directory of drive 1 of a multiple-drive disk. The CTRL key may be used to slow down the listing, and the STOP key may be used to terminate the directory command. DIR may ONLY be used in the interpretive mode (i.e. cannot be compiled into a definition).

- LISTING MULTIPLE SCREENS ON THE VIC-1525 PRINTER

To list more than one screen, the following word will provide proper page formatting on a VIC-1525 printer:

```
: LS  1+  SWAP  DO  I  LIST  CR  CR  CR  CR  LOOP  ;
```

To list screens 60-79 on the printer:

```
4  4  0  0  0  OPEN
4  CMD#
60  79  LS
4  CLOSE
```

When C64-FORTH first starts running, it maps (switches) out the BASIC ROMs, and assumes the top of usable memory to be CFFF. It first allocates 512 bytes below this point for use as RS-232 buffers, should they be needed. Just below this, 128 bytes is allocated as the user area. The user area is actually 64 16-bit storage locations, used to hold system parameters of the current C64-FORTH system. Entries in the FORTH dictionary called User Variables store their contents here, not in the body of the definition in the dictionary.

C64-FORTH then allocates block I/O buffers below the user area. Four buffers are immediately allocated. The number of buffers may be any number from 2 up to what space permits. Each buffer takes up 1028 bytes. The command BUFFERS saves any updated screens, multiplies the desired number of buffers by 1028, checks if there is enough room between HERE+200 and the bottom of the user area (LIMIT returns that address). If there is, the specified number of buffers is reserved, and FIRST shows the new bottom of the buffer area. If not, the number of buffers that there is room for is reserved.

```
8 BUFFERS      ( reserve 8 block buffers, if there is
                 enough room for them )
```

It is important to note something here - Since PAD always points to a location 68 bytes above the current location of HERE (i.e., the end of the dictionary), and the C64-FORTH string extensions use the area from PAD to PAD+512, then you cannot use the string extensions if you have the maximum number of block buffers allocated that space permits. Also, if there is not enough space to allocate the requested number of block buffers, the user is not informed of how many buffers actually were allocated. However, using the values in FIRST and LIMIT and the value of 1028 bytes per buffer, it is easy to calculate the actual number allocated.

MEMTOP is used to lower the effective top-of-memory of the FORTH system. This is helpful in reserving a section of RAM that FORTH will not touch except by commands from the user. The RS-232 buffers and the user area will be relocated just below the address specified, and MEMTOP will attempt to allocate 4 block buffers just below the user area. In deciding how far you can move the top-of-memory down, check the current top of the dictionary (returned by HERE), add 840 to it (200+512+128), and add the number of buffers desired times 1028. Subtract the result from the physical top of memory (D000 hex). This will give you the maximum number of bytes that you may move the top down.

Example:

HERE 840 + HEX U. 60F2 OK (HERE + 200 + 128 + 512)

404 2 * 60F2 + U. 68FA OK (we only need 2 buffers)

D000 68FA - U. 6706 OK

(6706 hex is approx 26.4K bytes free. We'll reserve just 16K bytes. This will leave 10K free for adding new definitions.)

D000 4000 - MEMTOP (MEMTOP executes COLD when done)
C64-FORTH

2 BUFFERS OK

(prove that everything is lower)

HEX LIMIT U. 8D80 OK

FIRST U. 8578 OK

What if at run-time you would like to reserve some space to use as a buffer, but you are not sure which RAM locations between the top of the dictionary and the bottom of the buffers are safe to use? The best way is to request one or more of the block buffers. The BUFFER command allocates a 1K byte block of memory and assigns it to the block number specified. The previous contents of the buffer, if updated, are written out to disk. The new block is NOT read in. This is not safe to do if your program will be reading in blocks (screens). But for application programs where no disk or cassette I/O is done except for data file reading or writing, all allocated buffers are free to use.

A block number is passed to the BUFFER command. It's best to pick a block number that is not configured on any I/O storage unit. 7FFF is a good buffer number to request since it's meaningless. Just don't use a negative number. BUFFER returns the address of the first location in the 1K of space assigned.

Example:
HEX 7FFF BUFFER U. C376 OK

To create a table, array, or machine code overlay and save it on disk for future use, obtain use of a buffer as described above, except specify the screen number you want it stored into on disk. Immediately after obtaining the buffer, execute the UPDATE command to have that buffer marked as updated. Fill it, then execute SAVE-BUFFERS to have it written out to disk. To read in the screen at some future time, specify the screen number and execute BLOCK. BLOCK calls BUFFER to request space to read in a screen. The screen is read in and BLOCK returns the address of the first byte in the block.

Example:
(save a 1K table of data onto disk screen #20)

```
20  BUFFER          ( request space for block #20 - addr returned)

( execute code here that fills buffer in memory )

UPDATE  SAVE-BUFFERS      ( write block out to disk screen #20 )
```

- BORDER, BKGRND, AND CHRCLR VARIABLES

When C64-FORTH is entered from BASIC or when either COLD or WARM is executed, the colors of the C64 border, background, and characters printed are set from the color codes in these variables. These may be changed at any time. The color codes must be in the range 0-15.

- RESTORE KEY

The RESTORE key on the C64 keyboard, when pressed with the STOP key, can be used to restart the FORTH system, in case it 'hangs' up. This is effectively a 'warm-start', meaning that definitions created since boot-up are not l

RESTORE will not work if the FORTH kernal itself has somehow been corrupted, or if the interrupts were turned off when setting up special graphics modes in the VIC-II chip. Data or return stack overflow can also corrupt the system to where recovery is impossible. Upon boot-up, C64-FORTH writes a short NMI handler into the RAM underneath the kernal ROM. This is to allow proper NMI handling via the RESTORE function even if high-res graphics are currently being drawn to the RAM underneath the kernal ROM, in which case the kernal ROM is temporarily mapped out. If the kernal ROM is mapped out at the instant of a NMI interrupt, then only a RESTORE key generated interrupt is recognized. A NMI interrupt from a RS-232 channel input or output operation is disregarded, therefore RS-232 channel I/O CANNOT be enabled during graphics writing to RAM underneath the kernal ROM.

- FUNCTION OF DV1, DV2, DV3, AND DV4

Up to four storage devices may be easily configured into the C64-FORTH system. Four variables exist that define the characteristics of storage devices available, named DV1, DV2, DV3, and DV4. The variables each contain 4-bytes of data. The first two bytes contain the number of screens that device may hold. The third byte in each specifies the I/O device number of that device (only 1 or 4-15 is valid as a device number). The fourth byte contains the drive number of the unit. Normally zero, this can be set to 1 to access drive #1 for 4040 dual-drive type disk units attached via an IEEE cartridge.

When the number of screens in one of the variables is zero, then that I/O unit, as well as the following units, is not available for storage of screens. The number of screens a device is allowed to store is found by reading the first 2-bytes of its variable:

```
DV1  @  U.      ( will print out the # of screens/blocks allowed
                  on memory device #1 )
```

You may change the number of screens allowed on a memory device (or to enable a new device if it is attached to either the serial bus or via an IEEE interface) by storing a new number in the device variable. For instance, you have just hooked up another 1541 drive (the internal jumpers were changed to configure it as device #9), and you want to use it to store screens. The first 1541, device #8, is characterized by DV1. We must now characterize DV2 to allow C64-FORTH to recognize the new drive. We must inform C64-FORTH of the number of screens, the bus device #, and the drive #:

```
120  DV2  !  9  DV2  2+  C!  0  DV2  3  +  C!
```

Each screen takes up 5 disk blocks. A screen takes up space on disk ONLY when a screen has been stored to the disk. But assuming the full number of allowable screens are stored on each disk, 120 is the maximum practical amount storable on a 1541. If you intend to have other programs stored an a screen disk at the same time, you must lower the number of allowable screens to compensate. For convenience, lets make sure DV1 is set to 100 (C64-FORTH is shipped with DV1 set to 100, but lets make sure):

```
100  DV1  !
```

And to make sure C64-FORTH doesn't think there are any more devices available, we will store a zero into DV3 and DV4:

```
0  DV3  !  0  DV4  !
```

Next we should set the device and drive number of the new drive:

```
9  DV2  2+  C!  0  DV2  3  +  C!
```

DV1 should already have 8 and 0 stored as the device # and drive #. What we have just done is configure the C64-FORTH system as follows:

- Memory device #1 is bus device #8, drive #0, and holds FORTH screens # 0 to 99. The screens are stored on the disk currently in drive 0 of device #8 as files with names SCR0 thru SCR99.

- Memory device #2 is bus device #9, drive #0, and holds FORTH screens # 100 to 219. The screens are stored on the disk currently in drive 0 of device #9 as files with names SCR0 thru SCR119.

Note that the file names of screens stored on a disk are relative to SCR0. This means that if, referring to the above configuration as an example, you store a screen as screen # 123, it will be saved on drive #0 of device #9 as SCR23. You can at a later time put the disk that was in device #9 into device #8, and access that same screen as screen #23. The way the algorithm works is:

- Screens are saved on memory device #2 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory device #1 (i.e. the contents of DV1 are subtracted from the requested screen #).

- Screens are saved on memory device #3 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory devices #1 and #2 (i.e. the contents of both DV1 and DV2 are subtracted from the requested screen #).

- Screens are saved on memory device #4 as files with names SCRxx where xx is the user-specified screen # minus the number of screens allowed on memory devices #1, 2, and 3 (i.e. the contents of DV1, DV2, and DV3 are subtracted from the requested screen #).

Commands such as LIST, EDIT, BLOCK, and COPY work on any screens on any configured disk, as long as the requested screen # is within range. If not, an error message will be displayed.

The cassette unit may be used as freely as a disk drive for storing screens. Now since the FORTH will return a screen full of blanks if a requested screen does not already exist on the disk, how does C64-FORTH know that a screen does not already exist on a cassette? There is no way of it knowing. A variable called TAPEFLG has been added for that purpose, and is described elsewhere in this section.

- FUNCTION OF SYSDEV#

SYSDEV# is a variable used by the C64-FORTH I/O routines. In any case where a BLOCK is read in or transferred out, R/W automatically sets this variable from the values in DV1, DV2, DV3, and DV4. But when a SAVESYSTEM or SAVETURNKEY is executed, the program is saved onto whatever device number happens to be in SYSDEV#. This is no problem if the disk drive that C64-FORTH was loaded from is the one you wish to save the current FORTH system to. If not, or if you've done block I/O (loading or editing screens for instance) from a different device, then you must set SYSDEV# before executing a SAVESYSTEM or SAVETURNKEY. The low byte of SYSDEV# holds the device #, and the high byte holds the drive #.

Example:

```
8  SYSDEV#  !  SAVESYSTEM  "CURRENT SYS"
```

- TAPEFLG

TAPEFLG is a variable used for cassette block I/O. The problem arises when you wish to edit a screen that does not already exist on the cassette. With the disk, no problem. R/W detects the FILE DOES NOT EXIST status from the disk, and creates a blank screen in memory to start with. But since the cassette cannot return that status, TAPEFLG was created to preserve the structure of the R/W routine.

If the screen number we wish to edit does not exist on cassette, the FORTH system could look forever for it and never find it. So when a screen is to be read in, and the device that screen is associated with is the datasette (i.e. device #=1), then TAPEFLG is checked. If TAPEFLG is =0, then the cassette is searched for the screen. If TAPEFLG >0 and <256, then an inquiry is printed on the screen asking whether it should look for the file on the tape, or should create a new blank screen in memory and assign it to the screen number specified. Any reply, including just hitting the RETURN key, will cause searching for the file on the cassette. Only by pressing the 'N' key will a blank screen be created.

- VALIDATING THE SCREENS DISK, AND DISK PECULIARITIES

FORTH screens are saved onto the disk as sequential data files. When a screen is to be saved out to the disk, the data is saved in the REPLACE mode. This means the filename the screen was saved out under was preceeded by "@0:". This causes no problems most of the time, but occasionally a nearly-full disk will act strangely. Periodic VALIDATEing of the disk (type DC "V0") plus switching to a new empty disk to hold additional screens when an old one becomes 80-90% full will prevent any problems.

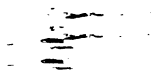
One other detail - the only time a Commodore disk drive knows that the floppy disk in it has been switched is by detection of a different disk ID#. When you NEW a disk, it is imperative you assign a unique 2-number or letter code for the ID# (this 2-digit code follows the disk name in the NEW command you sent the drive to format the disk). Never use the same 2-digit code for more than one disk. And it is a good habit, although not absolutely necessary, to initialize a disk that has just been swapped into the drive (DC "I0").

- ERROR MESSAGES

An error message is printed as a question mark ('?') followed by a message number or informative text. As long as the C64-FORTH definition named 'EMSGS' exists in the dictionary, the error messages printed will be fairly descriptive. Right after an error, executing WHERE will print out the line at fault and an arrow pointing to just after where the error occured. Look for missing spaces between FORTH words, looping and branching structures mispaired, values left on the stack will cause problems within looping and branching structures, etc. Sometimes just the message is printed as a warning (e.g. ISN'T UNIQUE or EMPTY STACK), and at other times control is returned to the FORTH input interpreter (as in USE ONLY WHEN LOADING). Screens 4 & 5 on the C64-FORTH disk contain copies of the error messages. In the glossary, refer to EMSGS and ERROR. If an error occured when defining a colon definition, and the definition cannot be FORGETten, execute SMUDGE. The most recent header can then be removed.

- USING IEEE INTERFACE CARTRIDGES

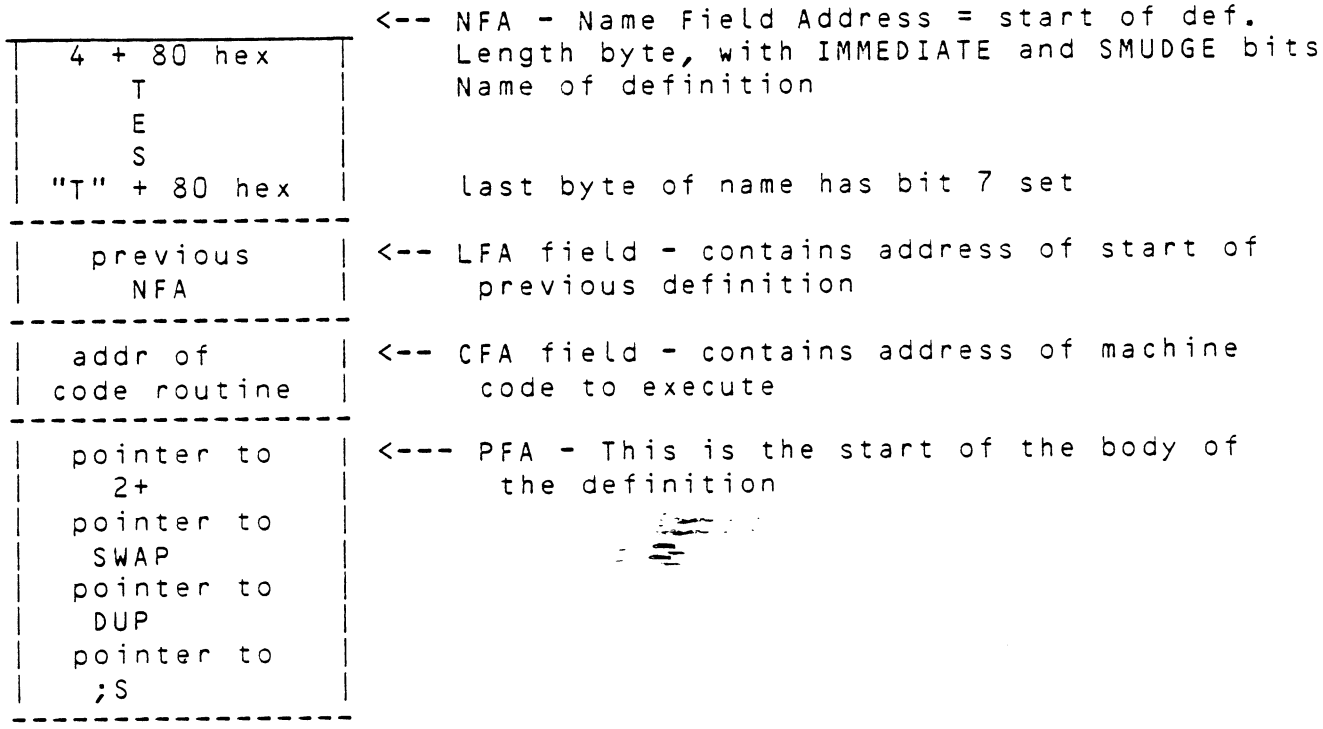
C64-FORTH will automatically work with any Commodore PET/CBM/VIC/C64 compatible disk drive. For the ones needing an IEEE interface, certain ones are available that are cartridge based and plug directly into the 64 cartridge slot. ROM code is a necessary part if any catridge-base interface. The internal 64 kernal I/O routines must be modified to divert commands and data from the serial bus to the IEEE interface. Since any cartridge ROM in the range 8000-9FFF is mapped out (made unaccessible as the RAM underneath is mapped in) when C64-FORTH maps out the BASIC ROMs, a way was provided to disable the disabling of the BASIC ROMs. Immediately after loading C64-FORTH, POKE 2067,255 before RUNning C64-FORTH. The BASIC ROMs, as well as any cartridge ROMs, will be left accessible. Only one thing to remember - there will be ROM code in the range 8000-BFFF once you enter C64-FORTH. Normally, the lowest disk buffer will overlap the BASIC ROM area. You must either decrease the number of buffers down to 2, or MEMTOP the system down to 8000 hex. MEMTOP is safer, since you won't have to worry about the end-of-the-dictionary overlapping the ROM code.



- THE STRUCTURE OF A DEFINITION IN MEMORY

We have been referring to different parts of a definition in memory. It is time to briefly cover the structure of a definition as it appears in memory. Lets show how the following definition will look in memory:

```
:  TEST  2+  SWAP  DUP  ;
```



The Name Field Address (NFA) is the address of the start of the header. Every definition works upwards in memory, so the NFA is the address of the lowest byte of that definition. The first byte of the header is used for various things. The lower 5 bits of it contain the count of the number of characters in the name. Bit 7 is always set to indicate the start of the header. Bit 6 is set to indicate if the definition is marked as IMMEDIATE, meaning it will execute immediately even within a new colon definition being created. Bit 5 is called the SMUDGE bit. When set, the definition will be listed with VLIST, but the definition cannot be executed, FORGOTten, etc. When : is used to create a new colon definition, the SMUDGE bit is set. Only when the terminating ; is excuted, does the SMUDGE bit get cleared. This prevents executing a colon definition that was not properly completed, due either to an error encountered within the source code of the colon definition, or the user just forgetting to terminate the definition with ; .

After the header byte, the name of the definition follows, with bit 7 being set in the last character of the name. After the name field comes the LINK field. This is a 16-bit pointer to the start of the previous definition (the NFA of the previous definition) WITHIN THE VOCABULARY IT WAS CREATED UNDER. This is what links all the definitions together for the purpose of vocabulary searches. The FORTH word LFA will provide the address of this field within this definition.

Next comes the CODE field. The code field contains a 16-bit pointer to a machine code routine. If this definition was defined as a colon definition, this will contain the address of a routine in FORTH that will sequence through the pointers contained in the body of this definition. If defined as a constant, the address will point to a routine that will push the 16-bit contents of the body onto the stack, and so on for other types of definitions. This Code Field Address points to this location.

And finally comes the body of the definition. The address of the first byte of this field is called the Parameter Field Address. If the definition is a colon definition, the body is made up of pointers to other word definitions. A colon definition terminates with a pointer to ;S (EXIT is the same thing). ;S tells the FORTH interpreter that the current definition is completed and to 'un-nest' one level - that is, return to the definition that called it.

If the definition is a variable or constant, then the body is only two bytes long, containing the current contents of the constant or variable.

- DEBUGGING FORTH ROUTINES USING TRACE FEATURE

A simple trace function is built into the system. It allows the user to single-step through the execution of a FORTH colon definition. Definitions composed of machine code cannot be stepped through. They will execute entirely in one step.

The trace feature is turned on by setting bits in location 132 (84 hex) of zero page memory. If bit 7 is set by writing 128 (80 hex) to this location, trace mode is immediately turned on. But since you have to go through a lot of FORTH kernal routines for the parsing and matching of input stream words, setting bit 6 is preferred for tracing one specific routine. There is a word in the FORTH dictionary which does that for you, called TON.

Example of usage:

```
3  7  TON  *
```

This will turn the trace mode on after the values 3 and 7 are pushed on the stack, and before * begins executing. You will skip over the FORTH system overhead, and tracing will begin just as the definition * is entered.

The name of the routine to be stepped through should follow immediately after turning on the trace; otherwise the end-of-the-line terminator, (the X <null> routine, which is actually defined as a word definition) will start tracing. Any parameters required by the routine to be examined should be put on the stack before trace is turned on.

Example:

```
( to trace the routine */ )
3  10  6  TON  */   ( turn on trace just before */ )
```

Two types of trace lines are shown. A normal line will look like this:

```
2031  */MOD  66  01F3  0006  000A  0003
```

All numbers are shown in HEX to provide a reasonable format on the 40-character C64 screen.

The first number displayed is the address within the current colon definition of the NEXT pointer to be fetched. The routine this pointer identifies will be the next to be executed, and the name of that routine follows the address.

The next number specifies that actual address of the bottom of the data stack (which is the TOP stack item). When the stack is empty, this value is 6C. It decrements by two for each data item on the stack (4 for each double number).

The return stack pointer follows. The return stack is used to hold the next pointer address when nesting down one level, DO loop index and limits, and temporary values placed on by >R. The return stack pointer also decrements by two for each entry placed on it.

On the remainder of the line, if any values are on the data stack, they will be displayed. Only the top four values are shown, with the top stack item being the leftmost.

Every time FORTH 'nests' down one level (starts executing a lower-level colon definition), TRACE will show this by printing a colon in column one of the display followed by the definition's name:

```
:  */MOD
```

'Unnesting' (returning to the higher-level definition) will not be shown, but can be recognized when ;S or EXIT is about to execute. After ;S or EXIT executes, the return stack pointer can be seen to increment by two showing the FORTH has unnested one level.

After each line is displayed, hitting just the RETURN key will single-step through the definition. If you wish to turn off trace mode and finish executing the definition at full speed, press R . To terminate execution of the routine, turn off trace mode, and return to the FORTH input prompt, press X .

There is one other feature of the trace mode. At the flashing cursor, you can press B as an option. B immediately executes a 6502 BRK (break) instruction. If you have a monitor wedge or cartridge installed, you will enter the monitor. Make note of the address printed upon entry to the monitor. You will need this address to return to C64-FORTH. You may now use the monitor to examine and modify memory locations, even to single-step (if it is a feature of your monitor of course!) through machine code routines if no FORTH system zero-page variables are disturbed. When you desire to return to FORTH, type G xxxx where xxxx is the address you wrote down when you entered the monitor. You will have executed the next FORTH word in the definition you were tracing, and you may continue to single-step through that definition. If the BRK vector has not been changed by a monitor prior to initial entry into C64-FORTH, then the FORTH redirects the BRK vector to a warm start routine, so the B command in trace does not crash when no monitor is present.

- MEMORY MAP

The COMMODORE-64 memory map, with C64-FORTH in control, looks like this (all addresses in HEX):

0000-00FF — 6502 page zero memory. Contains data stack, C64 I/O and system variables.

0100-01FF — 6502 return stack. Bottom 80 bytes used as FORTH terminal input buffer.

0200-03FF — C64 variables and buffers.

0400-07FF — Display memory.

0800-4xxx — C64-FORTH program. HERE leaves the address of the current end of the FORTH dictionary. PAD, the pointer to the text output buffer, always starts 68 bytes after the HERE address.

PAD+80 thru BD6F — Usable RAM for expanding the FORTH dictionary, machine code routines, work buffers, etc. Note that the C64-FORTH editor uses this area for cut-and-paste temporary storage.

BD70-CD7F — Block buffers. As shipped, C64-FORTH is setup for 4 buffers. If the number of buffers is changed, or MEMTOP is changed, then this address range will be different.

CD80-CDFF — User area. The user area is 128 bytes long, and is a collection of variables describing the characteristics of the current FORTH system. This address range will change if MEMTOP is changed.

CE00-CFFF — RS-232 buffers. If device #2 is OPENed, these buffers will be used by the 64 kernal ROM routines.

D000 — By default, the highest address +1 of usable continuous RAM. MEMTOP may be used to lower the effective top-of-memory as perceived by the FORTH system.

D000-DFFF — COMMODORE-64 I/O register space.

E000-FFFF — COMMODORE-64 I/O and screen display ROMs. For high-resolution bit-mapped graphics, the RAM that 'hides' behind these ROMs may be used.

The COMMODORE-64 BASIC ROMs are normally in the address range A000-BFFF, but are mapped out by C64-FORTH. They are replaced by usable RAM. The ROMs are re-enabled when exiting to BASIC.

SAVETURNKEY is the only way any version of C64-FORTH may be redistributed. It also has the advantage of making the application program more end-user oriented, since the FORTH system itself is no longer accessible. The program, after being loaded from BASIC and entered via RUN, initializes the system and immediately executes the top word in the FORTH vocabulary. Any error exits back into the FORTH kernal will restart that last definition. The RESTORE key also works properly and causes a restart.

The procedure for creating a turnkey system is as follows:

1) Write and debug your application program (enough said).

2) If desired, to shorten load time of the application program, you may forget the editor and error messages by:

   FORGET EDITOR

   You may also not need the floating-point extensions, which take up about 4K of space. FORGET FPOVER to recover that space too.

3) Load your application program onto the FORTH system.

4) Insert a pre-NEWed disk into the disk drive. Set SYSDEV# to the device # you wish the program to be saved to.

5) Type the word 'SAVETURNKEY' followed by the program name.

   Ex. SAVETURNKEY "SPACE WIDGETS"

6) SAVETURNKEY immediately asks whether you really do want to create a turnkey system, in case you were managing the clouds when you entered the command. Reply by pressing the 'Y' key if you weren't.

7) It responds by telling you to insert the next disk (pre-NEWed, of course), and to press any key. The program is saved on disk or tape, and the inquiry is repeated. Do this for as many disks or tapes as you wish to create. Since the FORTH system has been thoroughly corrupted, you'll have to reboot C64-FORTH when done.

THE EDITOR

As mentioned earlier, source code in a FORTH system is stored in a group of blocks stored on disk or tape called 'screens'. When you go to edit a screen, the FORTH system first looks to see if it is already in memory. If not, it is called from the disk or tape unit first. While you are editing a screen, the changes being made are kept only on the display, the screen in memory is not being changed at the same time. When you exit the editor, you have the options of discarding the changes you made, updating the screen in memory, updating and saving it out to disk, or updating, saving it, and 'loading' (compiling) it.

To edit a screen, type:

<scr#> EDIT

If the requested screen # is not in memory, the screen will be read in from disk or tape. The first 1000 characters of the FORTH screen will be displayed. The remaining 24 characters of the FORTH screen will not generally be missed, but if necessary access to them is possible. All editing is done on the portion of the screen displayed. Editing a screen is done in much the same way as editing lines of a BASIC program. Several additional editing features have been provided to make the process even easier. Full cursor movement over the displayed screen is possible, however you are prevented from 'scrolling-up' the screen by going off the bottom. The color-select keys (<ctrl>-0, <ctrl>-1, etc.), cursor movement keys, RVS-ON, RVS-OFF, and HOME keys work normally. CLR-HOME does not clear the screen but performs the same function as HOME.

- SCREEN LINE LENGTH

Each line of the displayed screen is normally considered to be 40 characters in length. Inserting and deleting characters when in the middle of a screen line will cause the shifting of characters on that line only with the following two exceptions:

1) Function key 1 (f1) sets the current line to a length of 80 characters (the last displayed line cannot be set to a length of 80 characters). After this key is pressed, and up to when the current row changes via a cursor movement, pressing <return>, etc. the current line length will stay at 80. Any movement of characters on the current line from inserting or deleting characters will also cause shifting of characters on the following line, since it will be considered an extension of the current line.

2)If you are using the <del> key to delete the previous character, and you are currently at the first character of a line, the current line is then considered an extension of the previous line effectively making it 80 characters in length. The current line will be shifted into the previous line as you delete characters.

- BUFFERS USED BY EDITOR

Certain functions of the editor create and use buffers in the C64-FORTH memory space. The buffers are created within the space from above PAD up to the bottom of the disk buffers. An 80-byte CHARACTER buffer is used by the DELETE-CHARACTER-AND-SAVE function, and a LINE buffer is used by the DELETE-LINE-AND-SAVE function.

- INSERT functions

There are several different ways to insert text into the screen. First there is the insert key (activated by <shift>-DEL) on the 64 keyboard referred to as INS. INS causes all text starting from the current cursor position to the end of the line to be shifted to the right and a blank space is inserted at the current position. Text at the end of the line is 'shifted-off' and lost. Next, a character INSERT MODE is turned on by the <ctrl>-I key. Any further typing of alpha-numeric characters will expand the current line starting from the current cursor position and will insert the character typed. The insert-character function can be turned off by the <ctrl>-0 key. There is also an INSERT-BLANK-LINE function. The function key f4 moves the rest of the screen down starting from the current line and inserts a blank line. As to what happens to the bottom line that got shifted off the screen, refer to the section 'THE 26th LINE'.

- LINE CLEAR functions

Two keys perform line clear functions. The <ctrl>-C key clears the current line and places the cursor at the start of the line. <ctrl>-V clears text on the current line starting at the current cursor position up through the last character of the line. Note that both of these functions work according to the currently set line length. Also, because of the 64 keyboard decoding, pressing the STOP key will function identically to <ctrl>-C in clearing the current line.

- DELETE functions

There are also several ways to delete text. The DEL key on the 64 keyboard deletes the character to the left of the current cursor position and shifts the rest of the line left one character position to take up the space. Spaces are added to the end of the line as it is shifted left. Note that a current line can be shifted into the previous line. The <ctrl>-D key deletes the current character and collapses the rest of the line to take up the space. Function key 3 (f3) deletes the current line. As with the INS key and the insert-character function, any text removed from the screen via the DEL key, the <ctrl>-D key, or f3 is lost forever. CUT and PASTE functions, described later, are used to delete and save characters and lines so that they may be moved to other locations in the current screen or to another screen.

- CUT and PASTE functions

Five functions are provided for 'cut and paste' operations. Four function keys, and one control key are used to provide: DELETE-CHARACTER-AND-SAVE (f7), DELETE-LINE-AND-SAVE (f5), UNDELETE-CHARACTER (f8), UNDELETE-LINE (f6), and 'POP'-A-LINE (<ctrl>-P). Characters and lines, as they are deleted, are saved off in temporary buffers. The f8 and f6 function keys pull out and insert into the current screen line the most recently saved characters and lines, if any were. The characters or lines are removed from the buffers as they are UNDELETED. The 'POP'-A-LINE key takes the most recently-deleted line from the line buffer and inserts it before the current line, but does not remove it from the line buffer. POP-A-LINE may be used to duplicate a line several times within one or more screens. The characters and lines saved in the temporary buffers may be passed to other screens AS LONG AS THE LENGTH OF THE FORTH DICTIONARY DOES NOT CHANGE! This is done by editing one screen, deleting characters and/or lines using the f7 and f5 keys, exiting the screen-editing mode, entering a second screen for editing, and using the f8 and/or f6 keys to insert characters and/or lines into the new screen. Between exiting the first screen and entering the editing mode of the second screen, no definitions may be added, deleted, or loaded, as this will cause loss of the saved characters and lines.

The character buffer in which characters deleted by f7 are saved in is 80 bytes (characters) long. When full, the delete-character function will stop working. The line-buffer uses any available space between the top of the FORTH dictionary and the bottom of the screen buffers area. Here too, when the space is filled, line-delete no longer works, and neither undelete-character nor undelete-line works when there are no characters or lines left in the buffers.

Note that the color of deleted characters and lines is not saved. Characters and lines are undeleted in the current color.

- THE 26th LINE

Where are the remaining 24 characters that aren't displayed? They reside in a 'mini-line' just below the bottom of the screen. Whenever any type of line-insert function is done, lines are pushed down the screen toward the bottom. The bottom line goes into the 26th line, the 'mini-line'. Whenever any type of line-delete function is done, lines are shifted up the screen and the 26th line is brought in and placed on line 25 of the screen. Since it is only 24 characters long, line 25 is padded with spaces on the screen. As line 26 is moved onto the screen, it is filled with spaces, although you can't see it. Also, when lines are shifted down the screen, the first 24 characters of line 25 go into line 26, the remaining 16 characters of line 25 are lost, as are the 24 characters that were in line 26.

Therefore, if you need all the space available in a screen, putting text in the last line is simple. Delete one of the lines currently on the screen using the DELETE-LINE function key (f5), which will save it in the line buffer. Move the cursor down to line 25 on the screen and type in the text that you want. Only the first 24 characters count. When set, move the cursor back up to where the line was deleted. Use the UNDELETE-LINE function key (f6) to fetch back the deleted line. The 24 character line at the bottom of the screen will disappear as lines are shifted down - it went into line 26. If you are worried about it being present, just use f5 again to delete lines and you'll see line 26 come up into the screen. Be careful inserting lines when you have text in line 26 as it will be shifted out and lost. Again, for most applications, screens are plentiful and to ignore the presence of the last 24 characters is no loss.

- NON-TEXT DATA FOUND IN A SOURCE SCREEN

The editor is for editing source text screens. Any non-text data found when the screen is first displayed is converted to a graphics character (usually the 'pi' character). When a screen is exited with update, the screen ASCII text is transferred into the screen buffer memory. Thus, if you create screens on disk which contain random byte data and not source text, and you accidentally enter the editing mode on one of those screens, you must exit WITHOUT UPDATE to preserve the data. The 64 has many control characters in it's display-control character set and allowing random byte data to be displayed would create a rather useless display. Hence, most non-text data bytes are displayed as a graphics character.

There are modes where control characters are accepted and displayed. Identical to BASIC, any control characters entered while in the `quote' mode (i.e. within paranthesis) inserts a graphics representation of the control character. Also, any color-select key pressed while not in the quote mode enables the key selected, and the <shift>-COMMODORE key switches the case (the display font) as in normal BASIC editing mode.

- EXITING THE EDITOR AND SAVING OR DISCARDING WORK

There are four ways to exit the editor. <ctrl>-Z exits, and discards all work done. The contents of the screen in the disk buffer memory area is not in any way modified or marked as updated. <ctrl>-X exits the editor, updates the contents of the screen in memory, and marks that screen as updated. <ctrl>-F exits, the editor, updates the screen in memory, and goes through the disk buffer area and writes out to disk ALL screens that are marked as updated, which will include the screen just edited. The numbers of the screens are printed as they are saved. <ctrl>-L exits the editor, updates the screen in memory, saves all updated screens to disk, and begins loading the screen just the same as if you had typed: <scr#> LOAD. With <ctrl>-L or <ctrl>-F, if for any reason there is a disk or tape error while FORTH is trying to save the screens, the screens will be left as updated in memory, the error message will be printed, and control will return to the FORTH input routine. The problem causing the error can be corrected, then the screen(s) can be saved again by the FORTH command SAVE-BUFFERS.

EDITOR COMMAND SUMMARY

f1 - set current line to effective length of 80
f2 - set current line back to effective length of 40
f3 - delete current line (line is not saved)
f4 - insert a blank line

f5 - delete current line and save
f6 - undelete a line and insert at current row
f7 - delete current character and save
f8 - undelete a character and insert at current pos

INS - insert a blank at the current cursor pos
DEL - delete previous character

<ctrl>-B - 'tab' (move cursor) 4 positions to the right
<ctrl>-C - clear current line (RUN/STOP key does same)
<ctrl>-D - delete current character (character not saved)
<ctrl>-I - turn on auto-insert
<ctrl>-O - turn off auto-insert
<ctrl>-P - 'pick' the most recent line in the deleted-line buffer and insert before current line
<ctrl>-V - clear to end-of-line

<ctrl>-Z - exit editor, discard all work done this screen
<ctrl>-X - exit editor, update screen in memory
<ctrl>-F - exit editor, update screen, and save screens
<ctrl>-L - exit editor, update screen, save all screens, and load this screen

THE ASSEMBLER

An assembler for generating 6502 machine code has been included in source code format with the C64-FORTH package. Once the assembler has been loaded into the FORTH dictionary, the entire FORTH program can be 'SAVESYSTEMed' and the assembler will be a permanent part of the bootable system. The assembler screens are screens # 6-17. To load the assembler, insert the C64-FORTH disk and type:

```
6 LOAD
```

Screen # 6, when through loading, automatically initiates loading of screen # 7, and so on until the entire assembler is created as part of the FORTH system. You may then save the new FORTH system onto a disk with the following command:

```
SAVESYSTEM "<filename>"
```

The assembler is provided in source code format because even though the 64 has approx. 51K of usable RAM when the BASIC ROMs are disabled, some C64-FORTH users may need every bit of available space. Once an application package has been written and debugged, the editor may no longer be needed. It consumes approx. 2.5K of memory, and it was placed as the last part of the C64-FORTH package. The editor may be deleted via 'FORGET EDITOR' freeing up the 2.5K. Since the assembler may be selectively loaded, the 2K of space that it consumes need not be taken up if the application package is written all in higher-level FORTH. Thus, you can customize your FORTH system with or without the editor, and with or without the assembler.

The assembler allows generating 6502 machine code directly from easily associated 'mnemonics', or instructions. Assembly language programs allow a one-to-one correspondence between program instructions and micro-processor operations. While FORTH produces very space-efficient and fast running code, some applications require the absolute control of the micro-processor that only machine code can give.

For users just picking up assembly language programming, the COMMODORE-64 PROGRAMMER'S REFERENCE GUIDE has a brief introduction on the subject, and there are several good books out on the market on programming the 6502 in assembly language. One useful point relative to C64-FORTH is that you don't need the 64MON cartridge for the 64 to write, enter, and execute machine language programs.

- CODING IN C64-FORTH ASSEMBLY

Coding in C64-FORTH assembly language is similar to coding in higher-level FORTH. Routines may originally be coded in FORTH and debugged, then recoded into assembly language with minimal changes or restructuring. Structure control directives (BEGIN---UNTIL, IF---ELSE---THEN, etc.), while implemented differently in the assembler vs. FORTH, function similarly. Therefore, to assure that assembler versions are used in assembly code, and to differentiate assembler mnemonics from some FORTH word definitions, all assembler control directives and mnemonics end with a comma (','). This also allows FORTH word definitions to be used during assembly to generate instruction addresses or to control the assembly process. For example:

```
CR ." ASSEMBLING BACKGROUND COLOR-SET ROUTINE"

  ( get current cursor color, and mask out high 4 bits
    using machine code AND instruction, not FORTH
    AND function )

646 LDA, HEX F # AND, DECIMAL          ( mask color value )
1 # CMP, 0= IF,                ( is current cursor color = 1? )
  0 # LDA, 53280 STA,      ( yes, set border color to black  )
  53281 STA,             ( and set background color to black )
ELSE,                      ( current cursor color not white, )
  1 # LDA, 53280 STA,      ( then set border color to white  )
  53281 STA,             ( and set background color to white )
THEN,                      ( terminate conditional branches  )
RTS,                       ( return from this subroutine     )
```

In traditional assembly language code, a label is optionally specified first, then the instruction mnemonic, followed by the 'operand' (the address to operate upon). In FORTH, to ensure proper structured coding, labels are not permitted. Operands must preceed the instruction mnemonic, since in FORTH values and parameters used as qualifiers must be put onto the data stack before the routine that operates on them is called, and instruction mnemonics are actually defined as word definitions.

Note in the previous example the use of the HEX and DECIMAL word defs. The assembler does not COMPILE definitions, but is actually operating in the INTERPRETIVE mode, just like FORTH does after it prints 'OK' on the screen and waits for user input. The assembly code instruction mnemonics are fed into the input stream just the same as if you were typing them in in response to the 'OK' prompt. Therefore, the FORTH control structures, such as BEGIN---UNTIL , IF---ELSE---THEN , etc. cannot be used since FORTH is not compiling a definition. Instruction mnemonics are FORTH routines that execute immediately upon being read in. They generate and place into memory byte data that will effect the proper instruction.

You will also note in the table of C64-FORTH assembler mnemonics that there are none of the 6502 'branch' instructions. In order to encourage good programming technique, control directives must be used to effect branching, such as BEGIN,---UNTIL, , IF,---ELSE,---THEN, etc. (Note that the ASSEMBLER versions have a comma at the end of the directives. It is important to be aware of this distinction.) The control directive limitation has another advantage - while assembling source code, the assembler checks control directives for proper nesting. This helps ensure that program branching during run-time execution is at least predictable. An alternative to this would be to have the FORTH kernal control the execution of each instruction, and this would be prohibitive timewise.

- ACCESSING MACHINE CODE ROUTINES FROM HIGHER-LEVEL FORTH

The instruction mnemonics and structure control directives are all defined in the ASSEMBLER vocabulary. The normal way to generate machine code routines is by using the FORTH word definition 'CODE'. CODE automatically creates a FORTH header, establishes the linkage into the machine code required for access from higher-level FORTH routines, and sets the active (context) vocabulary to ASSEMBLER. Instruction mnemonics can then be used to add machine code to the definition. Returning to the higher-level FORTH routines must be properly done or the system will be corrupted (i.e. expect it to crash!). Finally, the definition must be verified for proper completion. This is done with the 'END-CODE' word definition.

Machine instructions within a machine code routine cannot call routines defined with the CODE/END-CODE structure. To call a machine code routine from another machine code routine the 'JSR' (mnemonic for the call-routine function) instruction is used, and to return back to the next instruction of the caller the 'RTS' (mnemonic for return-to-caller function) instruction must be executed. This call-return sequence is different from how a FORTH level definition calls or invokes a sub-definition.
There is another way that a FORTH level definition can access (call) a machine code routine. The CALL and CALLR FORTH

definitions allow directly executing machine code routines that terminate execution with the RTS instruction. These machine code routines may then be ROM routines within the 64 memory space or code routines generated via the C64-FORTH assembler. And, of course, any machine code routine that ends with a RTS instruction can be called from any other machine code routine.

Various tests are made within the assembler to detect user errors. CODE saves the data stack level at the start of assembly language definition. END-CODE checks to make sure the exiting level is the same as the starting level detecting missing or excess parameter type of errors. Structure control directives place check digits on the stack to ensure proper nesting and pairing. Unimplemented addressing modes and out-of-range values for operands are checked for and flagged. While these measures don't guarantee crash-free code, they do show up many of the more common errors. A little user caution is also necessary in becoming familiar with the assembler. For instance, the FORTH definition 0= does something entirely different than the ASSEMBLER definition 0= even though it functionally appears the same.

There are a number of 6502 instructions that are called 'implied' instructions. There are no operands required for these instructions, and assembly of the mnemonic generates a one-byte data item in memory that is the machine instruction. The implied instructions are:

BRK,	CLC,	CLD,	CLI,	CLV,	DEX,
DEY,	INX,	INY,	NOP,	PHA,	PHP,
PLA,	PLP,	RTI,	RTS,	SEC,	SED,
SEI,	TAX,	TAY,	TSX,	TXA,	TXS,
TYA,					

In the list of instructions available under the assembler, branch instructions (other than JMP,) are not included, as these are generated automatically by the structure control directives. The group of instructions left are arithmetic, logical, and data movement instructions that have more than one addressing mode each. Not every one of the following instructions have the same allowable addressing modes, so check a 6502 instruction reference if you're not sure. The multiple-mode instructions are:

ADC,	AND,	ASL,	BIT,	CMP,	CPX,
CPY,	DEC,	EOR,	INC,	JMP,	JSR,
LDA,	LDX,	LDY,	LSR,	ORA,	ROL,
ROR,	SBC,	STA,	STX,	STY,	

(Actually, the JSR, instruction above has only one addressing mode, but since it is a memory reference instruction, we'll let it pass.)

The 6502 has 11 addressing modes for memory-reference instructions. The 6502 considers the bottom page (256 bytes) of memory to be special, calling it the 'zero-page' (zp), and offers shorter, quicker instructions for accessing those locations. You do not have to specify zp mode in assembler source code - the assembler checks to see if the effective address is in the zero-page, and if so, generates the special zp instruction instead of the general memory reference instruction. This works for indexed via X or Y modes as well as absolute references. Before the switch to zp mode is made however, the assembler checks to see if the zp mode is allowed. If not, the general memory reference instruction is generated anyway.

- ADDRESSING MODE SPECIFICATION

The assembler generates a one, two, or three-byte data item in memory that is the machine instrution plus the effective address. Simple symbols inserted after the effective address and before the instruction mnemonic conveys to the assembler which mode is desired:

SYMBOL	DESIRED ADDRESSING MODE	OPERAND RANGE
.A	accumulator	none
#	immediate data	byte value
,X	indexed,X	zp or absolute
,Y	indexed,Y	zp or absolute
X)	indexed via X, indirect	zp only (byte adr)
)Y	indirect, indexed via Y	zp only (byte adr)
()	jump indirect	absolute adr only
none	direct memory reference	zp or absolute

Examples:

FORTH ASSEMBLER	CONVENTIONAL ASSEMBLER SOURCE CODE AND COMMENTS	
66 ADC,	ADC 66	;ADD ZP LOCATION 66
FFD2 JSR,	JSR FFD2	;CALL ROUTINE @ FFD2
.A ASL,	ASL A	;SHIFT ACC LEFT
5F # CMP,	CMP #5F	;IS A = 5F ?
TABLE ,X LDA,	LDA TABLE,X	;FETCH TABLE ENTRY + X
50 ,Y CMP,	CMP 50,Y	;CHECK BYTE IN ZP BUFFER+Y
0 X) LDA,	LDA (0,X)	;FETCH VAL INDIRECTLY + X
50)Y SBC,	SBC (50),Y	;SUBTRACT INDIRECTLY + Y
FFFC () JMP,	JMP (FFFC)	;JUMP TO RESET ENTRY POINT

- RETURNING TO HIGHER-LEVEL FORTH

Returning to the FORTH system must be done properly or the system will crash. The re-entry point into FORTH is actually the interpretive-pointer sequencer. This routine executes the code pointed to by the next pointer in the body of a FORTH definition. The assembler assigns the name 'NEXT' to the entry-point of this routine. To properly exit a machine code routine and re-enter FORTH, execute a JMP to NEXT.

Example:

```
( turn off auto-key repeat )
CODE RPTOFF
     0 # LDA,          ( clear loc. 650 to disable )
     650 STA,
     NEXT JMP,         ( return to FORTH )
END-CODE
```

Values may be added or removed to the data stack at the same time that control is returned to FORTH by jumping to other re-entry points. All of these routines perform the indicated function, then jump to NEXT for you. Refer to ASSEMBLER reference list for detailed descriptions.

PUT	-Overwrite bottom 16-bit value on data stack
PUSH	-Add 16-bit value to bottom of data stack
PUSH0A	-Add byte value to stack as 16-bit value
POP	-Remove bottom 16-bit value from data stack
POPTWO	-Remove bottom 2 16-bit values from data stack

- THE DATA STACK

The FORTH data stack is accessable from assembly code. The data stack is a section of zero-page memory 80 bytes in length. The top of the stack when empty starts at 6C. If one word value is on the stack, the top of the stack is at 6A, with the low byte of the value accessable at loc 6A, the high byte of the value is at loc. 6B. Values can be added to the stack until it is full, i.e. when the data stack pointer gets down to 1C. Whenever control returns to the FORTH input interpreter, and the stack pointer is down below 20 (hex), a stack-filled error is issued.

Assembly code may freely access, remove, and add data to the data stack. The minimum data item that may be added to the stack is a 16-bit value. The stack pointer must always be decremented or incremented by an even number when adding or removing data items. The stack pointer is actually the 6502 X register. The zero-page indexed-via-X addressing mode is used to access values on the stack. For example:

```
0 ,X LDA,       ( fetch low byte of bottom of stack )
```

Since the stack starts from a certain memory location and works down as values are added, the current position pointed to by the stack pointer is referred to as the BOTTOM of the stack. The TOP stack data-item referred to in FORTH is actually at the bottom of the stack. Words have been defined in the assembler vocabulary to make references to positions of certain data-items on the stack. Since the low byte of the most recently added data item on the stack is accessed via addressing mode 0 ,X , then the assembler definition 'BOT' sets that addressing mode for the current instruction.

For example:

```
BOT LDA,
```

will fetch into the accumulator the low byte of the first value on the stack. BOT 1+ LDA, will fetch into the accumulator the high byte of the first value on the stack. BOT 2+ LDA, will fetch the low byte of the second value on the data stack, BOT 3 + LDA, will fetch the high byte of the second value, and so forth.

Since the second data item on the stack is also frequently needed by a code routine, the symbol SEC has been provided. Thus,

```
SEC STA,
```

will store the contents of the 6502 accumulator into the low byte of the second data item on the stack. SEC is therefore equivalent to BOT 2+ or to the address specification 2 ,X , and SEC 1+ is equivalent to BOT 3 + , etc.

- RETURN STACK

The C64-FORTH return stack is actually the 6502 machine stack. The 6502 uses page 1 of memory (from 100 to 1ff) for saving return addresses when subroutines are called. FORTH itself uses the return stack to save the interpretive pointer, to save the index and limit of DO loops, and to save temporary values. It is very easy to 'crash' the FORTH system and the 64 by improperly adding or removing values to the return stack. Only experienced FORTH programmers should attempt operations on the return stack (with the exception of some of the return operations to FORTH, as listed in RETURNING TO FORTH).

Values on the return stack may be obtained via the PLA instruction. The byte order convention is the same as with the data stack -i.e. low byte, then high byte. Therefore, to pull (remove) a 16-bit value from the return stack, PLA, TAY, PLA, will first remove the low byte of the value, save it in the Y register, then the high byte will be removed and saved in the accumulator.

To access values on the return stack without removing them, the S register of the 6502 is used. The S register is the machine stack pointer. It can be transferred into the X register, then any byte on the return stack may be accessed via the ,X addressing mode. Note that the X register must be saved and later restored before returing to FORTH. The process is:

1) save the X register in XSAVE

2) execute the TSX, instruction to bring the S reg to X

3) use 'RP)' as the address specification to address the lowest byte of the return stack. Use offsets to address higher bytes (RP) 1+ , RP) 2+ , RP) 3 + , etc.). The addressing mode is automatically set to ,X .

4) restore X before returning (load X from XSAVE)

The accumulator is not set to any particular value upon entry into a CODE routine. It may be freely used and does not have to be set to anything specific upon return to FORTH.

The X register is used as the data stack pointer by FORTH. If it is used for anything but accessing the data stack, it's value must first be saved, the operation performed, then it must be restored before returning to FORTH. Data items may be added to the stack by decrementing X twice for each item, or incrementing twice to remove each item.

Examples:

```
( push the value of 3 onto the data stack )
DEX, DEX,       ( make room for new entry  )
3 # LDA,        ( put 3 in low byte        )
BOT STA,
0 # LDA,        ( and clear high byte      )
BOT 1+ STA,

( set top of memory pointer from address on stack   )
BOT LDA, 643 STA,   ( fetch address and set pointer )
BOT 1+ LDA, 644 STA,
INX, INX,           ( remove item by incrementing X )

( set the screen cursor address from values passed on
  stack:    COL#  ROW#  ---
  NOTE: the X register is changed )
CODE SETCURSOR
  XSAVE STX,                          ( save X register )
  SEC LDY,                    ( set Y = desired col # )
  BOT LDA,                    ( set X = desired row # )
  TAX,
  CLC,              ( carry bit must be cleared to set )
  65520 JSR,                          ( call ROM routine )
  XSAVE LDX,              ( restore X before returning )
  POPTWO JMP,     ( remove values from stack & return )
END-CODE
```

The Y register is set equal to 0 upon entry into a CODE routine. It may be freely used and does not have to be restored before returning.

The S register, the 6502 machine stack pointer, points one byte below the low byte of the bottom return stack item. Improper alteration of the S register or values on the return stack will almost guarantee a system crash.

The 6502 is in binary mode upon entry to a CODE routine. If 6502 decimal calculations are done, binary mode must be reinvoked (via CLD,) before returning.

FORTH uses several variables while running to control execution of FORTH code. These variables are only accessable via machine code.

IP is the interpretive pointer. It points to the next pointer in the body of a FORTH word def. This will be the next word definition executed by FORTH when control is returned to it via jumping to NEXT.

W points to the code field of the FORTH definition just executed. W-1 contains the value 6C, which is the 6502 machine value for an indirect jump. FORTH enters a definition composed of machine code by pointing W to the code field address of the definition, which points to the machine code following it. FORTH jumps to W-1, the indirect jump instruction, the pointer is fetched out of the code field of the definition, and the 6502 is then executes the machine code of the definition.

UP is the User Pointer that contains the address of the base of the user area - a collection of variables defining the characteristics of the current FORTH system.

There also is a work space area used by FORTH that can be used by machine code routines. These are 9 consecutive bytes referenced by the assembler as N-1, N, N+1, ... N+7. These are zero-page memory locations, so they may be used as pointer locations for indirect addressing instructions, or as temporary storage locations. Traditionally N-1 is used for byte values, and N, N+2, N+4, and N+6 are used to hold 16-bit values. The 9 locations may be used freely. Values put into these locations may be passed to called subroutines, but not to other FORTH word definitions. Many of the routines in the FORTH kernal use the work space locations.

XSAVE is a byte location in zero-page memory that is used to save the X register during routines that need it for other purposes. Some of the examples in this section show how it is used.

- SETUP

SETUP is a subroutine in the FORTH kernal that is callable by a machine code routine. SETUP removes a specified number of 16-bit values from the data stack and stores them in the work area. To use the routine, the accumulator is loaded with the number of values to remove from the stack, then just call SETUP. The number of items to remove can only be from 1 to 4! The bottom stack item is put into N,N+1 , the second item is put into N+2,N+3 , etc.

Example:

```
3 # LDA, SETUP JSR,
```

In this example, the top 3 data-items are removed from the data stack and placed into N through N+5.

- CREATING MACHINE CODE SUBROUTINES

While within a CODE definition another CODE definition cannot be entered, a machine code subroutine that ends with the RTS, instruction can be called. CODE creates a definition header that is compatible with FORTH higher-level definitions, as well as setting up conditions that allow END-CODE to check for errors. We need something simpler just to create a header for a machine code routine. We can use 'CREATE' for this purpose. CREATE will create a header with the specified name. Assembly code may then be used to make up the body of the definition. The definition must end with either the RTS, instruction, or a JMP, to a machine code routine that terminates with the RTS, instruction. END-CODE is not needed to complete the definition, since no error-checking conditions were set up initially. When the name of the new definition is referred to, the ADDRESS of the first instruction in the body of the routine is placed on the data stack. Therefore, the NAME of the machine code routine may be used as an address specification in another machine code or CODE definition.

Example:

```
( create machine code routine first to return <> 0 if
  STOP key on 64 keyboard is depressed )
HEX
ASSEMBLER       ( select ASSEMBLER vocabulary since CODE
                  is not used )
CREATE STOPKEY?                                ( create header )
  DC01 LDA,                          ( read in I/O port key bits )
  7F # CMP,                          ( if STOP pressed, then =7f )
  RTS,              ( return with Z status bit set or cleared )

( routine callable from FORTH waits until stop key pressed )
CODE WAITSTOP
  BEGIN,                                      ( start of wait loop )
    STOPKEY? JSR,                           ( see if STOP pressed )
  0= NOT UNTIL,                       ( loop until returns <> 0 )
  NEXT JMP,                                   ( return to FORTH )
END-CODE
```

A note of caution: Make sure you properly terminate a machine code routine. Since no error checking is done, the system could crash if a routine is called that was not terminated with a RTS, instruction or equivalent JMP, instruction.

- BRANCHING AND LOOP CONTROL STRUCTURES

As in higher-level FORTH, control structure directives must be used for program branching. The assembler versions of the directives effect the proper branch by placing into memory byte data corresponding to the 6502 branch instructions. They also place onto the data stack 'check' digits to help detect improper nesting of structures or incomplete structures.

The control structure directives are used for conditional looping or branching. The conditions allowed for branching are:

CONDITIONAL SPECIFIER	CONDITION TESTED
CS	is 6502 carry bit set ?
CS NOT	is 6502 carry bit clear ?
0=	is 6502 zero flag set ?
0= NOT	is 6502 zero flag clear ?
0<	is 6502 minus flag set ?
0< NOT	is 6502 minus flag clear ?
VS	is 6502 overflow flag set ?
VS NOT	is 6502 overflow flag clear ?

When a conditional specifier is used, a value is pushed onto the stack at assembly time that allows the directive that follows to place into memory the proper branch instruction to effect the testing of the condition.

CONDITIONAL FORWARD BRANCHING:

```
<conditional specification> IF,...ELSE,...THEN,
                            IF,...THEN,
                        (or IF,...ELSE,...ENDIF,
                        and IF,...ENDIF, )
```

As in higher-level FORTH, this structure is used for conditional forward branching. The condition is placed before the IF, part. Byte data is placed in memory such that at runtime if the condition tests TRUE, then control falls through to the first instruction after the IF, part. If not, a branch is taken to the first instruction after the ELSE, part (or THEN, if no ELSE, part was specified).

What the assembler version IF, actually does is remove the conditional placed on the stack, compile a branch instruction and temporary branch offset into memory, then place the current dictionary address onto the stack along with a check digit (the number '2'to indicate an IF,ELSE,THEN, type of structure). The way the stack is altered during assembly is:

```
conditional  ---  IFaddress  2
```

Remember that this action is performed during the ASSEMBLY process, not during run-time. During run-time, either the branch is taken, or the code following the branch is executed.

If an ELSE, directive is encountered in the input text, then first the check digit is checked to see that it is a '2' indicating the innermost conditional structure was started by an IF,. The address is then removed and the proper branch offset is calculated and placed into the branch instruction generated by the IF,. The current dictionary address is then placed onto the stack, along with a new check digit (also = 2). A JMP, instruction is placed into memory to allow the code executed by the IF, part to jump around the ELSE, part. A temporary jump destination address is placed in the instruction. The effect of the ELSE, during assembly is:

```
IFaddr  2  ---  ELSEaddr  2
```

Assembly code is assembled into memory until a THEN, directive is encountered. THEN, first checks the check digit to ensure it is a '2', then corrects the JMP, destination address if an ELSE, was part of the control structure, or corrects the branch offset generated by the IF, directive. The effect of THEN, during assembly is:

```
IFaddr  2  ---          or    ELSEaddr  2  ---
```

CONDITIONAL LOOPING:

```
BEGIN, ... <conditional specification> UNTIL,
                                       (or END, )
```

This structure is used for conditional looping or backward branching. The BEGIN, is used to mark the start of the loop, or the destination of the backward branch. Assembly code follows, then a condition is specified followed by the UNTIL, (or END,). The effect of this structure is to loop until a specified condition tests TRUE. Control then falls through, and the code following the UNTIL, part is executed.

BEGIN, only pushes the current dictionary address onto the stack followed by a check digit equal to 1. The effect of BEGIN, during assembly is:

```
---  BEGINaddr  1
```

UNTIL, first checks the check digit on the stack to verify that the inner-most control structure is a BEGIN,UNTIL, type of structure. Then, using the conditional and the address placed on the stack by BEGIN, , it generates a backward branch instruction in memory whose destination is the first instruction given after the BEGIN, directive. The effect of UNTIL, during assembly is:

```
BEGINaddr  1  conditional  ---
```

CONDITIONAL EXECUTION AND LOOPING:

BEGIN, ... <conditional specification> WHILE, ... REPEAT,

This structure is used primarily for executing code within a loop until a condition is met. The BEGIN, is used to mark the start of the loop. Assembly code may optionally follow, then a specified condition code is tested for. If the condition tests TRUE , then the code between the WHILE, and REPEAT, parts is executed. The REPEAT, causes an unconitional branch back to the first instruction after the BEGIN,. If the condition tests FALSE, then the loop is exited and the first instruction specified after the REPEAT, is executed.

BEGIN, only pushes the current dictionary address onto the stack followed by a check digit equal to 1. The effect of the BEGIN, during assembly is:

--- BEGINaddr 1

When the WHILE, directive is encountered, first the most recent check digit on the stack is checked to see that it is a '1' indicating the innermost control structure was started by a BEGIN,. The equivalent to an IF, directive is executed to place a conditional forward branch instruction into memory, however a 4 is left on the stack as a check digit. The effect of the WHILE, during assembly is:

BEGINaddr 1 conditional --- BEGINaddr 1 WHILEaddr 4

When the REPEAT, directive is encountered, the second check digit on the stack is checked to insure it is a '1' indicating this whole mess was started with a BEGIN,. A JMP, instruction is placed into the dictionary at this point to jump back to the first instruction after the BEGIN, part. Next, the first check digit on the stack is checked to insure it is a '4' indicating a WHILE, was specified. The equivalent to a THEN, directive is executed to terminate the conditional forward branch instruction placed into memory by the WHILE, part. The effect of the REPEAT, during assembly is:

BEGINaddr 1 WHILEaddr 4 ---

Examples:

```
( Wait for a key to be pressed - if key pressed was 'L' then
  put a TRUE flag on the stack, else, put a FALSE flag on
  the stack. )
HEX
CODE ?LKEY
  XSAVE STX,                                  ( save X register )
  BEGIN,                                ( start of key-input loop )
    FFE4 JSR,                 ( get a value from keyboard buffer )
  0= NOT UNTIL,                                ( wait until <> 0 )
  4C # CMP,                                ( is 'L' key pressed ? )
  0= IF,
    1 # LDA,                          ( yes, put true flag in acc )
  ELSE,
    0 # LDA,                          ( no, put false flag in acc )
  THEN,
  XSAVE LDX,                                ( restore X register )
  PUSHOA JMP,       ( push flag onto stack & return to FORTH )
END-CODE


( move sprite #0 down the screen until it either reaches
  raster line # 150 or collides with another sprite )
HEX
CODE DOWNSCRN
N 1- STY,    ( use N-1 as terminating flag - zero it first )
BEGIN,                                          ( start of loop )
  D001 LDA,     ( is sprite #0 at or beyond line 150 yet ? )
  DECIMAL 150 # CMP, HEX
  CS IF,
    1 # LDA, N 1- STA,          ( yes, set flag to terminate )
  THEN,
  D01E LDA,    ( has sprite collided with another sprite ? )
  1 # AND,
  N 1- ORA,                            ( 'or' in terminate flag )
0= WHILE,                                 ( exit if result <> 0 )
  D001 INC,    ( else, increment row position of sprite #0 )
REPEAT,                                   ( loop back and continue )
NEXT JMP,                                        ( return to FORTH )
END-CODE
```

As referred to in the section BRANCHING AND LOOP CONTROL STRUCTURES, the assembler checks for proper nesting of control structures and terminates with an error condition if improper nesting is discovered. What if, for instance, you want to forward branch out of a BEGIN,...UNTIL, loop upon a certain condition? There is a way to 'trick' the assembler into generating the proper branch instructions by manipulating the check digits placed on the stack by the control directives.
In the above case, the control structure you want to implement is:

BEGIN, ... <conditional> IF, ... <conditional> UNTIL, THEN,

Following the previously described course of assembly for the directives, the condition of the data stack as the IF, directive is encountered is:

BEGINaddr 1 conditional --- BEGINaddr 1 IFaddr 2

Since the innermost structure is a IF,THEN, type of structure, and the next directive encountered is UNTIL, , you must trick the assembler into thinking the innermost structure is a BEGIN,UNTIL, type. Both the check digits AND the addresses placed onto the stack must be switched. 2SWAP works very nicely for this purpose. 2SWAP must be used AFTER the IF, , and before the conditional specification of the UNTIL, directive. After the 2SWAP is executed, the data stack will look like:

BEGINaddr 1 IFaddr 2 --- IFaddr 2 BEGINaddr 1

Then when the <conditional> UNTIL, is encountered, the proper backward branch is placed into memory, and when the THEN, is encountered, the conditional forward branch created by the IF, is properly terminated.

Example:

```
( move sprite #0 down one raster line at a time until it
  either hits another sprite or a background pattern )
HEX
CODE SPRITEDOWN
BEGIN,                                          ( start of loop )
  D001 INC,                  ( move sprite #0 down 1 raster line )
  D01E LDA,     ( has sprite #0 collided w/another sprite ? )
  1 # AND,
  0= IF,                           ( if yes, branch out of loop )
    D01F LDA,  ( no, has sprite collided with background ? )
    1 # AND,
2SWAP                        ( switch addresses & check digits )
0= NOT UNTIL,                                ( branch back if not )
THEN,                                ( terminate forward branch )
NEXT JMP,
END-CODE
```

So you've heard all your assembly-language programmer friends extol the virtues of using MACROS? And, you figure, if they can use them in their 'legitimate' programming, why can't you use them in FORTH ASSEMBLER? Go ahead, we're not stopping you. MACRO capability in FORTH happens not to be a special added feature, but an allowable function just by default.

MACROS in assembly programming are a way of defining a sequence of source code instructions to be generated just by specifying the name of the MACRO and perhaps a qualifier or two. Sounds similar to the way normal higher-level FORTH works? You got it.

Since the FORTH assembler is comprised of definitions that execute upon 'source' text being read in, you can create a new definition that causes several previously-defined definitions (e.g. assembler mnemonics) to execute just by specifying the new definition. Also, since the assembler words use the data stack to calculate the effective addresses of machine code instructions, parameters or qualifiers may be passed to the new definition to customize the generated code.

You'll be wanting use of macros only when using the assembler, so it's best to define macros as part of the assembler vocabulary. This way macro definitions won't be searched of listed when doing normal higher-level FORTH work. When you are done defining macros, select the vocabulary you wish to add new definitions to. The procedure is:

```
ASSEMBLER DEFINITIONS

< macros are defined >

FORTH DEFINITIONS

< new CODE definitions can be created via the assembler
  but placed in the FORTH vocabulary >
```

A common example of a commonly desired macro is to increment a 2-byte (16-bit) pointer. Since the 6502 is a 8-bit machine, to do this in normal FORTH assembly code, using N+2 as the pointer location for this example, is:

```
N 2+ INC, 0= IF, N 3 + INC, THEN,
```

This code sequence must be used every time a pointer is to be incremented in the assembly code program. A macro can be defined as such:

```
ASSEMBLER DEFINITIONS
: INC2,  DUP INC, 0= IF, ROT 1+ INC, THEN, ;
FORTH DEFINITIONS
```

Then within the source code, the macro is invoked by:

```
. . .  N 2+ INC2,  . . .
```

Analysis of the macro INC2, will show that the proper code will be generated. Other examples of macros are:

```
ASSEMBLER DEFINITIONS

( clear the accumulator      ---    )
: CLRA, 0 # LDA, ;

( push the location of a pointer onto stack    addr  ---  )
: PSHLOC, DUP 1+ LDA, DEX, BOT STA, LDA, DEX, BOT STA, ;

( decrement a 16-bit counter    addr  ---    )
: DEC2, DUP LDA, 0= IF, 3 PICK 1+ DEC, THEN, DEC, ;

( push a pointer onto the stack & return to FORTH
  addr  ---    )
: PSHNEXT, DUP LDA, PHA, 1+ LDA, PUSH JMP, ;

( the following does the same thing as the one just defined,
  but shows example of nesting of macros )
: PSHNEXT, PSHLOC, NEXT JMP, ;
```

#

Sets 'immediate' addressing mode for the next instruction.

)Y

Sets 'indirect indexed via Y' addressing mode for the next instruction.

,X

Sets 'indexed via X' addressing mode for the next instruction.

,Y

Sets 'indexed via Y' addressing mode for the next instruction.

.A

Sets the accumulator direct addressing mode for the next instruction.

0<

Conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'N' status bit is set as the result of some previous operation.

0=

Conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'Z' status bit is set as the result of some previous operation.

;CODE

Used as a way to conclude a colon definition with machine code to create a new type of FORTH level definition. The new type is defined as:

: <typename> ...FORTH defs... ;CODE ...assembly code...

;CODE stops compilation and terminates the definition <typename> by compiling (;CODE) into definition. The CONTEXT vocabulary is set to ASSEMBLER, and the following assembly code is added to the definition. Note: at least one FORTH definition must be called between <typename> and ;CODE.

When <typename> is later used to define a new word as in:

<typename> <defname>

the word <defname> will be created with its execution procedure given by the machine code following ;CODE. That is, when <defname> is executed, the machine code specified after the ;CODE of <typename> is executed. (;CODE is actually defined in the FORTH kernal. The assembler redefines it as:
: ;CODE [COMPILE] ASSEMBLER [COMPILE] ;CODE ;
IMMEDIATE
which will make the ASSEMBLER the CONTEXT vocabulary before invoking ;CODE)

ASSEMBLER

Make the ASSEMBLER vocabulary the CONTEXT vocabulary. It will be searched first as each word is encountered in the input stream for a match.

BEGIN,

A structure control directive thats marks the destination address for a backwards loop. Used with either the UNTIL, directive or the WHILE, and REPEAT, directives, as in:

BEGIN, ... <conditional> UNTIL, or
BEGIN, ... <conditional> WHILE, . . . REPEAT,

BOT

An assembler symbol that sets the addressing mode to access the bottom byte of the data stack. An positive offset may be added to this symbol to access other bytes on the data stack. Used as follows:

BOT LDA, BOT 1+ STA, BOT 5 + ADC, etc.

CODE

A defining word used in the form:

CODE <defname> . . . . . END-CODE

to create a dictionary entry for <defname> in the CURRENT vocabulary. <defname>'s code field contains the address of its parameter field. When <defname> is later executed, the machine code in this parameter field will execute. The CONTEXT vocabulary is made ASSEMBLER to make available the assembly code instruction mnemonics and directives.

CS

A conditional specifier used before a control directive. This one causes the assembly of a conditional branch instruction that at run-time will test if the 6502 'C' status bit is set as the result of some previous operation.

ELSE,

A structure control directive that marks the end of the code to be executed if a status condition tested TRUE , and marks the start of the code to be executed if that status condition tested FALSE. Used in the following way:

```
<conditional>  IF, . . . ELSE, . . . THEN,
```

END-CODE

Used to mark the end of a CODE definition. END-CODE checks Checks that stack position is same as when CODE was executed to check for imbalanced directives or parameters. Then the most recently defined CURRENT vocabulary definition is unsmudged making it available for execution from FORTH. END-CODE exits the ASSEMBLER making CONTEXT the same as CURRENT. This word was named ;C in early versions of FORTH.

IF,

A structure control directive that creates a forward branch instruction based on a 6502 status condition. At runtime, if the status tests TRUE, execution falls through to the following code. If it tests FALSE, the processor branches. Used as follows:

```
<conditional>  IF, . . . ELSE, . . . THEN,    or
<conditional>  IF, . . . THEN,
```

IP

Assembler symbol that specifies the address of the FORTH interpretive pointer. Used as follows:

```
IP LDA,   IP )Y LDA,    etc.
```

At run-time, the NEXT routine moves IP ahead within a colon-definition. Therefore, IP points just after the execution address being interpreted. If an in-line data structure has been compiled (e.g. a character string), indexing ahead by IP can access this data.

N

Assembler symbol that specifies the address of a 9-byte workspace in zero-page memory. Within CODE routines and machine code subroutines, free use of the workspace locations may be made over the range N-1 to N+7.

NEXT

Assembler symbol that specifies the address of the normal

re-entry point in FORTH from a CODE definition. All CODE definitions must return to NEXT or to a routine that returns to FORTH (PUSH , PUT , PUSHOA , POP , POPTWO).

NOT

During assembly, NOT reverses the previous conditional specification. When the following control directive is encountered, a conditional branch instruction is generated that tests if the specified 6502 status bit is clear. For example:

```
CS NOT IF,
```

will, at run-time, fall through to the following code if the carry status bit is cleared, or will branch if the carry bit is set.

POP

Assembler symbol that specifies the address of the re-entry point in FORTH that will remove and discard the bottom data stack entry and then jump to NEXT.

POPTWO

Assembler symbol that specifies the address of the re-entry point in FORTH that will remove and discard 2 data stack entries, then jump to NEXT.

PUSH

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, push a new data-item onto the data stack. The contents of the accumulator will be used as the high byte of the new data-item, and the bottom byte of the return stack will be fetched and used as the low byte of the new item. For example, to return to FORTH and at the same time push the contents of N, N+1, :

```
. . . N LDA, PHA, N 1+ LDA, PUSH JMP,  END-CODE
```

PUSH0A

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, push the contents of the accumulator onto the data stack, then jump to NEXT. The value in the accumulator is placed as the low byte of the new stack entry, the high byte of the new entry is set equal to 0. Example:

```
. . . 1 # LDA, PUSH0A JMP,  END-CODE
```

will return and push a value of 1 onto the data stack.

PUT

Assembler symbol that specifies the address of the re-entry point in FORTH that will, at run-time, replace the current bottom entry on the data stack with the contents of the accumulator and the bottom of the return stack. The accumulator is used as the high byte, the bottom byte on the return stack is fetched and used as the low byte. Example:

```
. . . N LDA, PHA, N 1+ LDA, PUT JMP,  END-CODE
```

REPEAT,
A structure control directive that creates an unconditional jump instruction to the address left on the stack by a BEGIN, directive. Used in the following way:

```
BEGIN, . . . <conditional> WHILE, . . . REPEAT,
```

RP)
Assembler symbol that specifies the addressing mode required to access the bottom byte of the return stack. For this to work at run-time, the return stack pointer (the S register) must have been transferred to the X register). See section called 'RETURN STACK'.

SEC
Identical to BOT 2+ . Assembler symbol that specifies the address and addressing mode necessary to access the low byte of the second entry on the data stack. Used as follows:

```
SEC LDA,     SEC 2+ STA,   etc.
```

THEN,
A structure control directive that terminates a forward branch instruction created by either an IF, or an ELSE, control directive. Used as follows:

```
<conditional>  IF, . . . ELSE, . . . THEN,    or
<conditional>  IF, . . . THEN,
```

UNTIL,
A structure control directive that creates a conditional branch instruction that will branch to the address defined with a BEGIN, directive if the tested condition is FALSE. Program execution falls through to the following code if the condition tests TRUE. Used with BEGIN, to create a program loop or backward branch. Used as follows:

```
BEGIN, . . .  <conditional specification> UNTIL,
```

UP
Assembler symbol that specifies the address of the pointer to the base of the USER AREA. The user area is a block of variables that define the characteristics of the current FORTH system. Used as follows:

```
UP LDA,   UP )Y LDA,  etc.
```

W

Assembler symbol that specifies the address of the pointer to the code field (execution address) of the FORTH definition begin executed. Indexing from the W pointer can access any byte in the definition's parameter field. For example:

```
2 # LDY, W )Y LDA,
```

will fetch the first byte of the parameter field of the CODE definition executing these instructions.

WHILE,

A structure control directive that creates a forward branch instruction based on a 6502 status condition. Used between the BEGIN, and REPEAT, directives, WHILE, allows a way to exit a program loop. The branch is taken if the tested condition is FALSE. If it tests TRUE, execution falls through to the next instruction specified after the WHILE,. Used as follows:

```
BEGIN, . . . <conditional> WHILE, . . . REPEAT,
```

X)

Sets 'indexed via X, indirect' addressing mode for the next instruction.

XSAVE

Assembler symbol that specifies the address of a zero page location that can be used for temporarily saving the contents of the X register. Since the X register indexes to the data stack in zero page, it must be saved and restored when used for other purposes. Used as follows:

```
XSAVE STX, . . . XSAVE LDX,
```

FORTH GRAPHICS ON THE COMMODORE 64

The graphics extensions avaiable with C64-FORTH are not part of the normal FORTH system, but are an already compiled object code block loaded in and linked into the system. LIST screen 1 to see which screen to start loading to do this. There will be two options for loading the graphics - one will load them after the floating-point routines in the base FORTH system, the other alternative is to load them OVER the floating-point routines. If your program will not need real number capability, then you can save space by placing the graphics where the f.p. routines were. Note that to add the graphics extensions to the system, you must start with an unexpanded copy of the FORTH just as it boots up from the original C64-FORTH disk. You cannot add the graphics extensions if you have made ANY changes to the original FORTH system. Once the graphics extensions are added, you may then extend or customize the FORTH system to your needs, do a SAVESYSTEM to save it as a new system, etc.

To use the graphics extensions, you MUST be familiar with how the different graphics modes of the Commodore 64 work. A thorough reading of the section on graphics in the Commodore 64 Programmer's Reference Guide is a must. The following Graphics Glossary contains description of all the usefull words in the graphics extensions. You will also find several graphics demo programs on the C64-FORTH disk. Analyzing these will help you understand the function of the differenct words.

A bit-map display may be set up in either hi-res or multi-color mode. All the pixel-modifying routines, such as LINE, ARC, BOX, etc. work according to which mode is currently set. All the user has to do is select or deselect multi-color mode via the SETMC or CLRMC commands.

VIDMEM must be used to initialize the VIC-chip to display in certain memory ranges. If the character mode is selected, then the addresses of the character ROM and the screen memory must be specified. Since the VIC-chip operates in one of four 'banks' (16K ranges), the highest two bits of the address passed for the character ROM are used to set the bank #. Thats why in examples you'll see the routine to reset the VIC-chip to normal operation as:

```
HEX  1000  400  VIDMEM
```

Its easy to understand why 400 is passed as the address of the screen memory, but you may wonder why 1000 is passed as the address of the character ROM, when it is at absolute address D000 underneath the address space of the VIC-chip. The reason is that the VIC-chip is normally set to bank 0, so the two highest bits of the character ROM address must be 0's.

VIDMEM also sets two variables - MBASE and SBASE. MBASE is used by some routines to obtain the base address of the bit-map memory, so a specific pixel location can be calculated. SBASE is used to hold the address of the base of the screen memory in character mode, or the color memory in bit-map mode.

The bit-map graphics words draw within a specified window. The drawing window is defined by the variables, XLOW, YLOW, XLIM, and YLIM. The largest size of the window may be 0,0 to 319,199 (159,199 in multi-color mode). If the variables are set so that the window extends outside this range, this range is used as the window.

SETBM and SETMC both initialize these variables, so if you wish to redefine the window, you must do it after using these words to define the initial graphics modes. The order in which words whould be executed then to set up a graphics screen is:

VIDMEM, SETBM/CLRBM, SETMC/CLRMC, then alter the window variables XLOW, YLOW, XLIM, and YLIM.

Once the display is set up, you may switch between different display screens by using just VIDMEM. The other words are not needed once the VIC-chip is properly set. You only need VIDMEM to change the addresses that point to display areas.

The RAM underneath the C64 kernal ROMs may be used as freely for bit-mapped graphics as any other RAM. The pixel-modifying routines automatically switch in & out the ROMs as needed. The STOP/RESTORE key function will work properly even if the kernal ROMs happen to be switched out at the time of the keys being pressed, but RS-232 input or output happening coincidently may be lost or garbled.

GRAPHICS GLOSSARY

The following glossary contains descriptions of the graphics functions provided. It follows the same format as the FORTH glossary at the end of this manual. None of the words are part of any FORTH standard, and only the words that are of possible use to the programmer are described.

Most of the FORTH level graphics words merely call subroutines written in assembly language. The machine code equivalents of the higher-level FORTH words have the same name preceeded by an asterisk. In this glossary, if a FORTH level word merely calls a machine code subroutine, the name of the subroutine will given in parenthesis at the end of the description of the word. Assembly language programmers may program graphics with the same ease as in higher-level FORTH.

ARC Strtangle Endangle Radius ---

Draw an ARC or CIRCLE from starting angle specified to ending angle specified, with the radius specified. The center of the ARC or CIRCLE drawn is about the pixel value found in CURX,CURY. ARC drawing is always done counter-clockwise. A circle is drawn by specifying the starting angle equal to the ending angle, or the ending angle 360 degrees greater than the starting angle. The range for the angles is 0 <= angle <= 360 degrees. (*ARC)

Example of a 3/4 arc of radius 50 is:

```
0  270  50  ARC
```

Example of a full circle of radius 90 is:

```
0  0  90  ARC    or    0  360  90  ARC
```

BCLR X Y ---

Turn on clear-bit mode and write the pixel. (*BCLR)

BOX DeltX Delty ---

Draw a box starting from the current location in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+Delty. (*BOX)

BSET X Y ---

Turn on set-bit mode and write the pixel. (*BSET)

BXOR X Y ---

Turn on exclusive-or mode and write the pixel. (*BXOR)

CBASE --- adr

A variable that contains the base address of the character ROM. Normally set to D000, the user may change this if the character set has been redefined in RAM. Used only by RCHR, the routine that writes a ROM character into the hi-res screen.

CLRBM ---

Turns off bit-map mode.

CLRBMM ---

Clears the bit-map memory.

CLRMC ---

Turns off multi-color mode.

COSINE ang --- cosine

Returns a signed 16-bit integer that is the cosine of the angle specified. The angle specified should be in the range 0 <= angle <= 360 degrees. (*COSINE)

CURX --- adr

A variable that the user sets with the X value of the pixel to start drawing at (in the case of LINETO, RLINETO, BOX, or RECT) or the X value of the center of the arc or circle to draw. CURX will contain the X value of the last pixel drawn upon return from LINE, LINETO, RLINETO, BOX, or RECT, or if ARC is called, will contain the X value of the center as specified when the ARC routine was called.

CURY --- adr

A variable that the user sets with the Y value of the pixel to start drawing at (in the case of LINETO, RLINETO, BOX, or RECT) or the Y value of the center of the arc or circle to draw. CURY will contain the Y value of the last pixel drawn upon return from LINE, LINETO, RLINETO, BOX, or RECT, or if ARC is called, will contain the Y value of the center as specified when the ARC routine was called.

FILBMM colorval ---

Fills the bit-map memory with the 8-bit value passed.

FILCLR colorval ---

Fills the color memory with the 4-bit value passed.

FILSCRN colorval ---
Fills the screen memory with the 8-bit (2-nybble) value passed.

GETADR --- X Y
The current values in CURX and CURY are fetched and pushed onto the stack. (*GETADR)

GETCEN --- X Y
GETCEN returns on the stack the pixel address of the center of the current drawing window. (*GETCEN)

GFLGS --- adr
Variable that is used as flag bits to indicate various graphics modes. Can be ignored when using FORTH level graphics utilities, but the machine language programmer must set/clear certain bits for the pixel-writing routines to work properly. Bit 15 is set if multi-color pixel writing is desired, cleared for normal hi-res pixel writing. Bit 7 is set to have the pixel-writing routine write in set-bit mode, Bit 6 is set to have it write in clear-bit mode, and both bits 6 & 7 must be cleared to have it write in the exclusive-or mode. All the graphics routines that write to the multi-color or hi-res screen call one pixel-writing routine, and these bits must be properly set for the desired writing mode.

GRCLR ---
Turn on clear-bit mode. All subsequent pixel drawing will be done in clear-bit mode.

GRSET ---
Turn on set-bit mode. All subsequent pixel drawing will be done in set-bit mode.

GRXOR ---
Turn on exclusive-or mode. All subsequent pixel drawing will be done in exclusive-or mode.

LASTX --- adr
A variable that is updated when a pixel in memory is actually modified. When a pixel-writing routine (such as LINE) returns, LASTX will contain the X value of the last pixel that was written. For instance, if a line is being drawn that extends out of the current drawing window, CURX will return with the X value of the ending pixel of the line, but LASTX will contain the X value of the last pixel that was drawn within the current drawing window.

LASTY --- adr

A variable that is updated when a pixel in memory is actually modified. When a pixel-writing routine (such as LINE) returns, LASTY will contain the Y value of the last pixel that was written. For instance, if a line is being drawn that extends out of the current drawing window, CURY will return with the Y value of the ending pixel of the line, but LASTY will contain the Y value of the last pixel that was drawn within the current drawing window.

LINE strtX strtY endX endY ---

Draw a line from strtX,strtY to endX,endY. CURX and CURY left at endX,endY. (*LINE)

LINETO endX endY ---

Draw a line from the current pixel location in CURX,CURY to endX,endY. CURX,CURY updated to endX,endY. (*LINETO)

MBASE --- adr

Variable set by VIDMEM to point to the starting address of the bit-map memory.

MCCLRVAL N --- where 0 <= N <= 3

Sets the 2-bit value to write to multi-color bit-map memory when in clear-bit mode.

NOTE: If exclusive-or drawing function is selected (via GRXOR) in multi-color mode, then the pixel-modifying algorithm checks first to see if the 2-bit pair at the current pixel address is equivalent to the 2-bit value passed via MCSETVAL. If so, the 2-bit value passed via MCCLRVAL is placed in the current pixel location; otherwise, the 2-bit value passed via MCSETVAL is placed in the current location.

MCSETVAL N --- where 0 <= N <= 3

Sets the 2-bit value to write to multicolor memory when in clear-bit mode (see note under MCCLRVAL).

PIXMSK X Y --- adr msk

For a given X,Y location, PIXMSK returns on the stack the address of the byte in the bit-map memory that contains the pixel, and the mask that can be used on the byte, if read, to isolate the pixel value itself. (*PIXMSK)

RCHR chr# X Y ---

Draws a ROM character into the bit-mapped memory with the upper-left corner of the character starting at X,Y. The character is zoomed (enlarged) according to the current value in the variable ZOOM. Chr# is the same value as if you were POKEing a character into the screen memory - i.e. if you wanted a capital "A", chr# would equal 1. There are tables of the screen display codes in both the Commodore 64 User's Guide and the Commodore 64 Programmer's Reference Guide. (*RCHR)

RDPIX X Y --- N

Returns the current value of a pixel. If the screen is in hi-res mode, the value N returned will be in the range 0-1, if the screen is in multi-color mode, the value returned is in the range 0-3.

RECT Deltx Delty ---

Draw a filled-box starting from the current location in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+Delty. (*RECT)

RLINETO Deltx Delty ---

Draws a line in relative mode from the current pixel address in CURX,CURY to CURX+DeltX,CURY+DeltY. CURX,CURY is left at CURX+DeltX,CURY+DeltY. (*RLINETO)

RSPXY Sprite# DeltX DeltY ---

Moves the specified sprite a RELATIVE amount. DeltX is added to the current X position of the sprite, and Delty is added to the current Y position of the sprite. (*RSPXY)

SBASE --- adr

Variable set by VIDMEM to point to the starting address of the screen memory.

SETADR X Y ---

Sets the current address. Takes the top two stack values and places them in CURX,CURY. (*SETADR)

SETBM ---

Turns on bit-map mode.

SETBMM ---

Sets all bits in the bit-map memory high.
(eq. to 255 FILBMM)

SETMC ---

Turns on multi-color mode.

SINE ang --- sine

Returns a signed 16-bit integer that is the sine of the angle specified. The angle specified should be in the range 0 <= angle <= 360 degrees. (*SINE)

SP1X SP# ---

Sets the specified sprite # to 1X magnification (non-expanded in the X direction).

SP1Y SP# ---

Sets the specified sprite # to 1Y magnitication (non-expanded in the Y direction).

SP2X SP# ---

Sets the specified sprite # to 2X magnification (expanded in the X direction).

SP2Y SP# ---

Sets the specified sprite # to 2Y magnification (expanded in the Y direction).

SP<BKGD SP# ---

Sets the priority of the specified sprite to be less than the background.

SP>BKGD SP# ---

Sets the priority of the specified sprite to be greater than the background.

SPCOLOR SP# COLOR ---

Sets the sprite specified to the color specified.

SPDEFINE --- (at compile time)
--- adr (when created def is executed)

SPDEFINE is used to create a new dictionary entry, read in 63 consecutive ASCII representations of numbers, converts them to binary form, and stores them in the new definition. This word is provided for defining sprites. When the name of the new definition is later executed, the address of the first byte of the sprite data is pushed onto the stack. The word SPMOV may be used to move the data up to an accessible area in display memory. When either HEX or BINARY (2 BASE !) mode is used, this is a simple way to convert a graph-paper representation of a sprite figure into binary code. The graphic demos on the C64-FORTH disk contain some examples of how to use this word to define sprites.

SPMCOFF SP# ---

Turns off multi-color mode for the specified sprite.

SPMCON SP# ---

Turns on multi-color mode for the specified sprite.

SPMOV fromadr toadr ---

Moves a 63 byte chunk of data from the address 'fromadr' to address 'toadr'. The 'toadr' is rounded down by SPMOV to the nearest 64-byte boundary before the data is moved. Intended for moving a sprite definition from the FORTH dictionary up to the a memory range in the current video display bank.

SPMSK SP# --- mask

When passed a sprite # in the range 0-7 a mask is returned that corresponds to the binary power of the sprite #. This means that a sprite # of 0 returns 1, a sprite # of 1 returns 2, etc. The mask may be used to mask out unused bits in a VIC-chip register to look for something related to just one sprite. For example, to check if sprite #3 has collided with the screen background data, execute:

```
3 SPMSK D01F C@ AND
```

and a non-zero flag will be left on the stack if a collision occured. To check if any of sprites 0, 4, and 7 have collided with the background, execute:

```
0 SPMSK 4 SPMSK OR 7 SPMSK OR D01F C@ AND
```

SPOFF SP# ---

Turns off the specified sprite.

SPON SP# ---

Turns on the specified sprite.

SPPTR SP# adr ---

Set the sprite pointer at the end of the screen memory that corresponds to the specified sprite #. The address specified (adr) is converted to a one-byte sprite block address that points to a 64-byte block within the current 16K display bank. You must move the sprite data itself to the current display bank, starting on a 64-byte boundary. The starting address of the 64-byte boundary you moved the sprite data to is what you pass to this routine to identify to the VIC-chip where the sprite data is in memory.

SPXY SP# X Y ---

Set the displayed location of the specified sprite # to X,Y.

UD* d1 d2 --- dlow dhigh

Unsigned 32-bit by 32-bit multiply leaving a 64-bit product. Useful for scaling graphics objects. (*UD*)

VIDMEM adr1 adr2 ---

Sets up the VIC-chip to certain display addresses. Adr1 is the address that the VIC-chip should expect to find the bit-map/character memory at, and adr2 is the address the VIC-chip should expect to find the screen memory at. VIDMEM automatically rounds down both addresses to the proper boundaries. For example, the bit-map/character memory must start on a 2K boundary, and the screen memory must start on a 1K boundary. Once adr1 and adr2 are rounded down, MBASE is set to adr1, and SBASE is set to adr2, and the VIC-chip starts displaying from those locations.

For example:

E000 C400 VIDMEM places the hi-res bit-map memory in the range E000-FFFF, and the screen memory in the range C400-C7FF.

1000 400 VIDMEM (hex) resets the VIC-chip to normal display ranges (same as power up - with screen memory in range 400-7FF, and the ROM character memory is addressed in the range 1000-17FF).

XLIM --- adr

Variable used to specify the X value of the lower right corner of the current display window. Pixel modification will not be done when the X value is greater than this value. SETBM sets this to 319, and SETMC sets it to 159. To define a drawing window, change this value after SETBM or SETMC is used.

XLOW --- adr

Variable used to specify the X value of the upper left corner of the current display window. Pixel modification will not be done when the X value is less than this value. SETBM and SETMC both set this to 0. To define a drawing window, change this value after SETBM or SETMC is used.

XSCALE --- adr

Variable used to set the scaling factor for LINE, ARC, BOX, and RECT drawing. If the variable is set = 0, then no scaling is applied. For scaling along the X axis, set this with a value in the range 1-255. A value of 1 gives maximum scaling, 128 gives 50% scaling, and 255 gives minimum scaling.

YLIM --- adr

Variable used to specify the Y value of the lower right corner of the current display window. Pixel modification will not be done when the Y value is greater than this value. SETBM and SETMC both set this to 199. To define a drawing window, change this value after SETBM or SETMC is used.

YLOW --- adr

Variable used to specify the Y value of the upper left corner of the current display window. Pixel modification will not be done when the Y value is less than this value. SETBM and SETMC both set this to 0. To define a drawing window, change this value after SETBM or SETMC is used.

YSCALE --- adr

Variable used to set the scaling factor for LINE, ARC, BOX, and RECT drawing. If the variable is set = 0, then no scaling is applied. For scaling along the Y axis, set this with a value in the range 1-255. A value of 1 gives maximum scaling, 128 gives 50% scaling, and 255 gives minimum scaling.

0 XSCALE ! 230 YSCALE ! will give approximately round circles and arcs on most monitors.

ZOOM --- adr

Variable used to hold the ZOOM (enlargement) factor for writing ROM characters into the bit-map memory. Used only by RCHR. When set = 0, the character is drawn normal size, when set equal to 1, the character is drawn twice normal size, and so on up to a zoom factor of 15.

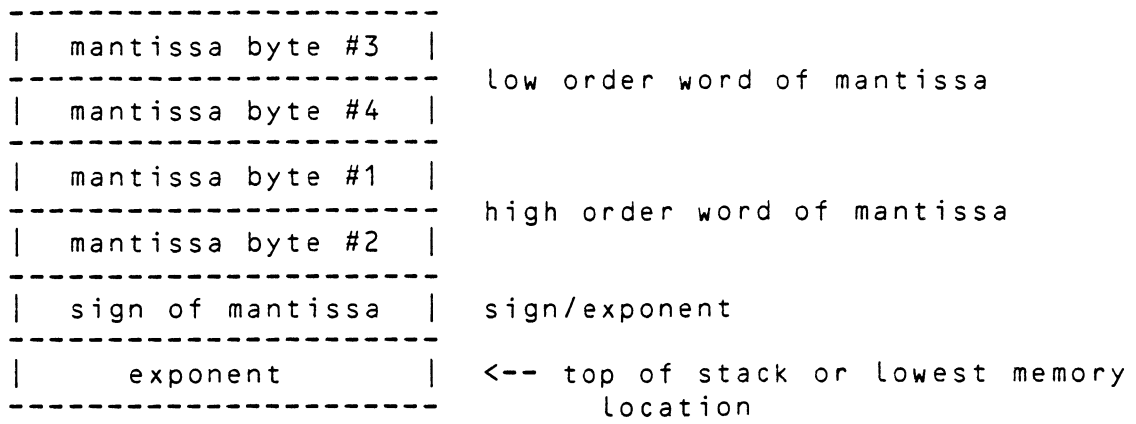
FLOATING POINT (REAL) NUMBERS

C64-FORTH comes with full floating-point capability. All of the functions available in BASIC for manipulating real numbers are provided in C64-FORTH. The floating-point words are part of the FORTH system as it exists upon boot-up. The graphics words follow the floating-point words, and the editor vocabulary follows the graphics. You would not want the editor to be part of any SAVETURNKEY package, since it is useless at run-time and takes up 2K or so of memory. You will want to FORGET EDITOR before loading in your application program. If you also do not want any of the floating-point commands, you may FORGET 79-STANDARD before loading in your application program to start with the smallest possible FORTH kernal. But what if you want graphics commands, but not floating-point? No problem, you may FORGET FPOVER to remove everything down through the floating-point routines. Then you may load screen 50. Screen 50 has a routine that will load in a pre-compiled version of the graphics commands that will start at the former starting location of the floating-point routines. Therefore you will have graphics commands available, but not the floating-point or editor commands taking up space. Remember that FORGETting any part of the boot-up FORTH system removes the ASCII error messages (see description of ERROR in the main gloassary). If you wish to re-add the error messages, load in screen 49.

Back to the floating-point functions. The floating-point commands are implemented mostly as CODE and assembly language routines. The routines are similiar to the floating-point code found in the Commodore-64 BASIC ROMs. There will be very slight accuracy differences between BASIC and FORTH functions, since using the FORTH stack as the main parameter-passing mechanism necessitates more frequent rounding-off than is done in BASIC.

C64-FORTH stores floating-point numbers in a slightly different format than BASIC does internally. While BASIC uses 5 bytes to store a f.p. number, C64-FORTH uses 6. This has two advantages - the FORTH stack will always change by an even number of bytes, allowing normal FORTH words to manipulate f.p. numbers on the stack, and the 6-byte format allows very easy conversion to the IEEE floating-point standard format, interfacing to an arithmetic processor such as the 9511, etc. C64-FORTH uses the same amount of precision as BASIC (7-bit signed exponent in excess-128 format, a 24-bit signed mantissa, and an 8-bit low-order accumulator during arithmetic operations, but again the mantissa is rounded up more often than in BASIC). The allowable range of floating-point numbers is the same as BASIC: +1.7E+38 thru +2.9E-39.

C64-FORTH stores floating-point numbers on the stack or in memory in the following format:



Bit 15 of the high-order word of the mantissa is always set upon return from the floating-point routines, but does not have to be set upon entering the routines. The sign of the mantissa is placed in bit 15 of the sign/exponent word, bits 8-14 are ignored, and bits 0-7 hold the exponent of the number. For instance, the number 1234.5678E+9 can be pushed onto the stack as follows:

```
HEX  FCD2 FB8 A9
```

Again, bit-15 of the high word of the mantissa can be set, so

FCD2 8FB8 A9 represents the same number. You may verify this by printing the number out using FP. :

```
FP.  1.2345678E+12  ok
```

Setting bit-15 of the sign/exponent word makes the number negative:

```
FCD2  FB8  80A9  FP.  -1.2345678E+12  ok
```

If the floating-point number on the top of the stack is equal to zero, the exponent will be zero. This is the only guaranteed indication of a value of zero! The sign bit may be set, and the mantissa may contain a non-zero value!

You don't normally have to worry about how floating-point numbers get onto the stack or the format they are in, since words are provided for converting integers and ASCII strings into floating-point numbers. But knowing the format allows you to use a few tricks, such as quickly testing whether the top floating-point number on the stack is negative, you can do:

```
.... DUP  0<  IF  ....
```

To test if the top stack f.p. value is =0, do:

```
.... DUP  FF  AND  0=  IF  ...  or  DUP  )LO  0=  IF
```

None of these methods remove the number in the process of testing it. They are also much faster than:

```
     ...  FPDUP  FP  0  FP<  IF  ...
```

or

```
     ...  FPDUP  FP  0  FP=  IF  ...
```

The f.p. commands are pretty simple to use, and work similiar to the integer FORTH functions. UNDERFLOW and OVERFLOW conditions are detected and handled as errors. Also, FPVAL takes the address of a string from the stack and converts the string to a floating-point number. The address of this string may be obtained using the C64-FORTH string extensions. Therefore, in BASIC you might have:

```
20 INPUT "ENTER ANGLE";A
```

While in C64-FORTH using string extensions:

```
16  VARIABLE  A$
: GETA  ." ENTER ANGLE? " A$ INPUT$ A$ FPVAL ;
```

If a print-right-justified floating-point equivalent of .R is desired, FPSTR$ converts a f.p. stack value into a string and leaves it at PAD. The string extensions may then be used to right-align the characters in a field of spaces.

(FP) --- fp

Not to be used by the programmer. This is the code routine compiled into a definition by FP . At runtime, (FP) takes the floating-point number that was compiled right after it and pushes it onto the stack.

DINTFP d --- fp

DINTFP converts the 32-bit integer d into a floating-point value. D is considered to be a signed value.

FP --- (at compile time)
--- fp (at run-time)

FP is used to input a floating-point value. FP executes whether it is within a definition being compiled or in the interpretive mode. FP scans the following input stream for an ASCII representation of a floating-point number. If the compile mode is not active, FP just leaves the f.p. number on the stack. If a definition is being compiled, FP compiles (FP) into the definition, then the value of the floating-point number. When the definition is later executed, (FP) fetches the f.p. value following it, and pushes it on the stack:

```
FP  -7.9E+12     pushes the value onto the stack

:  TESTFP FP -7.9E+12 FP. ;  will create a definition
called TESTFP that when executed will print the value
-7.9E+12.
```

FP! fp adr ---

Similiar to ! but stores a floating-point value into a 6-byte memory location. The location to store the value must be at least 6-bytes in length, and is normally reserved by using FPVARIABLE.

FP* fp1 fp2 --- fpprod

FP* multiplies fp1 by fp2 leaving the product fpprod.

FP+ fp1 fp2 --- fpsum

FP+ adds fp2 to fp1 leaving the sum fpsum.

FP- fp1 fp2 --- fpdif

FP- subtracts fp2 from fp1 leaving the difference fpdif.

FP.

fp ---

FP. takes the floating-point value off the stack, converts it to a string starting at PAD, then prints it. The value is printed with either a leading space or minus sign, and is followed by one space.

FP/

fp1 fp2 --- fpquot

FP/ divides fp2 into fp1 leaving the quotient fpquot.

FP<

fp1 fp2 --- flag

FP< leaves a true flag if fp1 is less than fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP=

fp1 fp2 --- flag

FP= leaves a true flag if fp1 is equal to fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP>

fp1 fp2 --- flag

FP> leaves a true flag if fp1 is greater than fp2, false flag if not. Both f.p. values are removed in the process of testing.

FP@

adr --- fp

FP@ fetches the 6-byte floating-point number starting at the address specified and leaves it on the stack.

FPABS

fp --- fpabs

FPABS returns the absolute value of a floating-point number.

FPADR

--- adr

A variable used by FPVAL. FPVAL is given the starting address of a string that is to be converted to a floating-point number. Conversion stops at the first non-valid character. FPVAL places the address of this character in FPADR. When FPVAL exits, the program can continue to scan off characters from the string by starting at the address left in FPADR.

FPATN

fp --- fpatn

FPATN returns the arctangent of the floating-point number passed to it. The result is the angle (in radians) whose tangent is the value fp. The result is always in the range -PI/2 to +PI/2.

FPCONSTANT fp --- (at compile time)
            --- fp (at run-time)

FPCONSTANT is used to create a constant in the dictionary that when referenced will push the floating-point value fp onto the stack. Can be used as follows:

```
FP  -7.56E9  FPCONSTANT  SCALEFAC

SCALEFAC  FP.  -7.56E+09  ok
```

FPCOS fp --- fpcos

FPCOS returns the cosine of the angle fp (the angle fp is assumed to be in radians).

FPDINT fp --- d

FPDINT converts a floating-point number into a double-integer. Note that the result d is valid only when the floating-point value passed is within the range of a double-integer.

FPDROP fp ---

FPDROP removes and discards the top floating-point value.

FPDUP fp --- fp fp

FPDUP duplicates the top floating-point value.

FPEXP fp --- fpexp

FPEXP calculates the constant 'e' (2.71828183) raised to the power of the value fp. A value of fp greater than 88.0296919 causes an OVERFLOW error.

FPINT fp --- n

FPDINT converts a floating-point number into a 16-bit integer. Note that the result n is valid only when the floating-point value passed is within the range of a single-precision integer.

FPLOG fp --- fplog

FPLOG returns the natural logarithm (log to the base of e) of the value fp. If fp is zero or negative, a "F.P. VALUE ILLEGAL" error occurs.

FPOVER fp1 fp2 --- fp1 fp2 fp1

FPOVER copies the second floating-point value on the stack and pushes it on top of the stack.

FPPI --- fppi

FPPI is a floating-point constant containing the value for PI (3.141592654). When referenced, FPPI pushes this value onto the stack.

FPRND n --- fprnd

FPRND returns a random floating-point value in the range 0.0 to 1.0. The integer n passed is used to determine the method of producing random values. If n is equal to zero, then FPRND generates a number directly from the system "jiffy clock". N being negative will cause a new "seed" value to be selected. If n is positive, the same "pseudorandom" sequence of numbers is returned, starting from the last selected "seed" value (done via a previous FPRND with n=-1).

FPROT fp1 fp2 fp3 --- fp2 fp3 fp1

The floating-point equivalent of ROT. Rotates the position of the top three floating-point values.

FPSGN fp --- n

FPSGN checks the value fp passed, and returns integer n. N is zero if fp was equal to zero. N is equal to 1 if the value fp was positive, and N returns equal to -1 if fp was negative.

FPSIN fp --- fpsin

FPSIN returns the sine of the angle fp (the angle fp is assumed to be in radians).

FPSQR fp --- fpsqrt

FPSQR returns the square root of the value fp. If fp is negative, a "F.P. VALUE ILLEGAL" error occurs.

FPSTR$ fp ---

FPSTR$ converts the floating-point value fp into an ASCII character string. The resulting string is left starting at the current value of PAD, and is compatible with all of the string extensions. The string has either a leading negative sign or space character depending on the value of fp.

FPSWAP fp1 fp2 --- fp2 fp1

FPSWAP swaps the top two floating-point stack values.

FPTAN fp --- fptan

FPTAN returns the tangent (in radians) of the value of fp. If the FPTAN conversion overflows, a "F.P. DIVIDE-BY-ZERO ERROR" occurs.

FPVAL $adr --- fp

FPVAL converts the string starting at $adr to a floating-point number. Conversion stops at the first non-valid character (decimal digits, ".", "E", "+", and "-" are valid characters). The address of the non-valid character that terminated the conversion is left in FPADR. A "F.P. OVERFLOW ERROR" will occur if the string represents a value out-of-range for f.p. values. Note that FPVAL will also terminate if the end-of-the-string is reached (FPADR will then point to the first byte after the string, which will not be meaningful data).

FPVARIABLE --- (at compile time)
--- adr (at run-time)

FPVARIABLE is used to create a 6-byte data area that can be used to store a floating-point variable. When the variable is referenced, the address of the data contained will be pushed on the stack. When the variable is created, its contents are not pre-initialized. FPVARIABLE can be used as follows:

```
FPVARIABLE  LVALUE

FP -7.56E9 LVALUE FP! LVALUE FP@ FP.  -7.56E+09  ok
```

FP↑ fp1 fp2 --- fprslt

The exponentiation function in BASIC is referenced by using the up-arrow (↑). C64-FORTH prefixes FP onto the name to remind you the function requires floating-point values. Fp1 is raised to the power of fp2 with the result being left on the stack.

INTFP n --- fp

INTFP converts the 16-bit integer n into a floating-point value. N is considered to be a signed integer.

UDINTFP ud --- fp

UDINTFP converts the 32-bit integer ud into a floating-point value. Ud is considered to be an unsigned value.

UINTFP u --- fp

UINTFP converts the 16-bit integer u into a floating-point value. U is considered to be an unsigned value.

C64-FORTH GLOSSARY

This glossary contains all of the word definitions in the kernal of C64-FORTH. The definitions are presented in the order of their ASCII sort.

The first line of each entry shows a symbolic description of the action of the procedure on the data stack. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes (---) indicate the execution point of the definition. Any parameters left on the data stack are listed to the right of the dashes. In this notation, the top of the stack is to the right.

The symbols include:

addr	memory address
b	8-bit byte (hi 8 bits = 0)
d	32-bit signed double integer (most significant portion with sign on top of stack)
f	boolean flag (0 = false, <>0 = true. True is usually = 1)
ff	boolean false flag (i.e. 0 is placed on stack)
n	16-bit signed integer (-32768 <= n <= 32767)
tf	boolean true flag (i.e. non-zero number is placed on stack)
u	16-bit unsigned integer (0 <= u <= 65535)
ud	32-bit unsigned integer (0 <= ud <= 4,294,967,295)

The following symbols are used in some of the definitions:

SYS	For advanced FORTH programmers only. Used internally by the FORTH system itself and is not a definition normally used by a FORTH programmer.
C64	Definition is not a part of the standard FIG-FORTH or FORTH-79 standards. This definition exists as either a necessary part of this implementation or as a convenience for the user.
79	This definition is part of the 79-std minimum required word set. Compatibility between different FORTH systems can be guaranteed when only words in this set are used.

The words in quotes (e.g. "store" or "tick") are standard pronunciations for some FORTH words.

! 79 n addr --- "store"
Store n at address specified.

!CSP "store CSP"
Save the stack position (data stack pointer) in CSP. Used as part of the compiler security.

79 d1 --- d2 "sharp"
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 .

#> 79 d --- addr count "sharp greater"
Terminates numeric output conversion br dropping d, leaving the text address and character count suitable for TYPE .

#S 79 d1 --- d2 "sharp s"
Generates ascii text in the text output buffer, by the use of # , until a zero double number d2 results. Used between <# and #> .

' 79 --- addr "tick"
Used in the form:
' <defname>
Leaves the parameter field address of dictionary word <defname>. As 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.

(79 "paren"
Used in the form:
(<text>)
Ignore a comment that will be delimited by a right parenthesis. May occur during interpreter input or within a colon definition. A blank must be present after the leading parenthesis before the start of the text.

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

(;CODE) SYS
The run-time proceedure, 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) SYS n ---
The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(-FIND) SYS addr ---
Called internally by -FIND and FORGET - The following word in the input stream is scanned off and a match in in the vocabulary pointed to by addr is attempted.

(/LOOP) SYS n ---
The run-time procedure compiled by /LOOP.

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

(DIR) C64
Called by DIR.

(DO) SYS
The run-time proceedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) SYS addr1 addr2 --- pfa b tf (if found)
addr1 addr2 --- ff (if not found)
Searches the dictionary starting at the name field 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) SYS 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) SYS
The run-time procedure compiled by LOOP which increments the loop index and tests for loop completion. See LOOP

(NUMBER) SYS 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 convertable digit. Used by NUMBER.

(R/W) C64 SYS
Called by R/W .

)HI C64 n1 --- n2
Take the high byte of n1 and leave as n2. Useful as a divide by 256 bit shift, or to generate a byte value out of an address for use by the assembler.

)LO C64 n1 --- n2
Leave the low byte of n1 as n2, zeroing out the high byte. Equivalent to 255 AND except faster. Useful for generating a byte value out of an address for use by the assembler.

* 79 n1 n2 --- n4 "times"
Leave the signed product of two signed numbers.

*/ 79 n1 n2 n3 --- n4 "times divide"
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 79 n1 n2 n3 --- n4 n5 "times divide mod"
Leave the quotient n5 and remainder n4 of the operation n1*n2/n3 . A 31 bit intermediate product is used as for */ .

+ 79 n1 n2 --- sum "plus"
Leave the sum of n1+n2.

+! 79 n addr --- "plus-store"
Add n to the value at the address.

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

+BUF SYS addr1 --- 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 SYS 79 n1 --- (run-time)
addr2 n2 --- (compile time) " plus-loop"
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). 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 SYS n --- addr
Leave the memory address relative by n to the origin parameter area. n is the minimum address unit, either byte or word. This definition is used to access or modify the boot-up parameters at the origin area.

, 79 n --- "comma"
Store n into the next available dictionary memory cell, advancing the dictionary pointer.

- 79 n1 n2 --- diff "subtract"
Leave the difference of n1-n2.

--> "next-screen"
Continue interpretation with the next disc screen.

-FIND SYS --- pfa b tf (found)
--- ff (not found)
Accepts the next text word (delimited by blanks) in the output strean to HERE, and searches the CONTEXT and then CURRENT vocabularies for a matching entry. If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-IN SYS --- addr
Called internally by ." and others to return the address of the next byte in the input stream.

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

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

." 79 ---
Used in the form:
." <text string>"
Compiles an in-line string <text string> (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 (.").

.LINE SYS 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.

/ 79 n1 n2 --- quot
Leave the signed quotient of n1/n2. The remainder has the sign of the dividend.

/LOOP n ---
Similiar to +LOOP except works with unsigned 16-bit numbers. Useful for using absolute addresses as the index and limit of a DO loop. N may be either a positive or negative increment. See +LOOP.

/MOD 79 n1 n2 --- rem quot "divide mod"
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 is attractive to define them by name in the dictionary as constants.

0< 79 n --- f "zero less"
Leave a true flag if the number is less than zero (negative), otherwise leave a false flag.

0= 79 n --- f "zero equals"
Leave a true flag if the number is equal to zero, otherwise leave a false flag.

0> 79 n --- f
Leaves a true flag if n is greater than zero.

OBRANCH SYS f ---
Never used by the programmer.
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+ 79 n1 --- n2 "one plus"
Increment n1 by 1.

1- 79 n1 --- n2
Decrement n1 by one.

2! C64 d1 addr ---
Store the double number d1 at addr. The high 16-bits of d1 are stored at addr, the low 16-bits at addr+2.

2* C64 n1 --- n2
Multiply top stack value by two. This routine uses a binary shift and is far faster than 2 * . Useful for bit shifting.

2+ 79 n1 --- n2 "two plus"
Increment n1 by 2. Same as 2 + .

2- 79 n1 --- n2
Decrement n1 by 2. Same as 2 - .

2/ C64 n1 --- n2
Divide top stack value by two. This routine is much faster than 2 / . Useful for bit shifting.

2@ C64 addr --- d1
Fetch the double number stored at addr.

2CONSTANT C64 d1 ---
A defining word similar to CONSTANT except for double numbers.

2DROP C64 d1 ---
Remove and discard the double number on top of the stack.

2DUP C64 d1 --- d1 d1
Duplicate the double number on the top of the stack.

2OVER C64 d1 d2 --- d1 d2 d1
Copy the second double number on the stack, placing it as the new top.

2ROT C64 d1 d2 d3 --- d2 d3 d1
Rotate the top three double numbers on the stack, bringing the third to the top.

2SWAP C64 d1 d2 --- d2 d1
Exchange the top two double numbers on the stack.

2VARIABLE C64 d1 ---
A defining word similar to VARIABLE but for double numbers.

79-STANDARD 79
A word which does nothing, however if the user can execute the word without an error condition occuring, the user can be assured all FORTH-79 extensions are present. For compatibility between different FORTH systems.

: 79 "colon"
Used to create a "colon definition" in the following way:

: <defname> . . . ;

Creates a dictionary entry defining <defname> as equivalent to the following sequence of FORTH words until the next ';' or ';CODE' . This word may then be used in new definitions or executed just by typing in '<defname>'. The definition must be properly terminated (with the ';' or ';CODE') and no errors should have occured. If not properly terminated, the name <defname> will be listed during a VLIST , however, the definition cannot be executed.

Other details are that the CONTEXT vocabulary is set to the CURRENT vocabulary and words used in the body of the definition that have their precedence bit set (i.e. were themselves defined as IMMEDIATE words) are executed rather than being compiled.

; 79
Trusted companion to colon (':'). Used to terminate a new colon-definition, unsmudge the header of the definition which makes it executable, and turns off the compilation mode. ; compiles the run-time word ';S' onto the end of the definition.

;CODE SYS
Used as a way to conclude a colon definition with machine code to create a new type of FORTH level definition. The new type is defined as:
: <typename> ..FORTH defs.. ;CODE ...assembly code..
Since the assembler is needed to add assembly code onto the FORTH definition, this word is more explicitly defined in the ASSEMBLER vocabulary.

;S
Used by the programmer within source code being loaded from a FORTH screen. Stops interpretation of the screen allowing the rest of the screen to be used for comments.

< 79 n1 n2 --- f "less than"
Leave a true flag if n1 is less than n2 , otherwise leave a false flag.

<# 79
Setup for numeric output formatting using the words:
<# SIGN #>
The conversion is done on a double number, producing text at PAD.

<CMOVE addr1 addr2 count ---
Identical to CMOVE except the highest byte is moved first, proceeding towards low memory. Useful for when the destination string will overlap the source string. See CMOVE.

= 79 n1 n2 --- f "equals"
Leave a true flag if n1 = n2; otherwise leave a false flag.

> 79 n1 n2 --- f "greater than"
Leave a true flag if n1 is greater than n2; otherwise leave a false flag.

>IN SYS 79 --- addr
A user variable containing the byte offset within the current input text buffer (terminal or block) from which the next text will be accepted. WORD uses and moves the value in IN.

>R SYS 79 n1 --- "to r"
Remove a number from the data stack and push it onto the return stack. Use should be balanced with R> in the same definition.

? 79 addr --- "question mark"
Print the value contained at the address in free format according to the current base (same as ' @ . ').

?COMP SYS
Issue error message if not in compile mode.

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

?DISK C64 SYS --- f
Checks the variable SYSDEV# and returns a true flag if the I/O device number found there is a disk (dev# > 3); otherwise returns a false flag. Used by (R/W) for forming filename, checking status, etc.

?DUP 79 n1 --- n1 (if zero) "query dupe"
n1 --- n1 n1 (non-zero)
Duplicate 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.

?ERROR SYS f n ---
Issue error message number n if the flag on the stack is true.

?EXEC SYS
Issue an error message if not in execution mode.

?FILE C64 --- f
Returns flag true if file exists on disk. First checks the I/O status byte and aborts if error. Then, if the system device is a disk (i.e. SYSDEV# >=4), ?FILE reads the disk status and checks if file exists. (R/W) calls this after attempting an open on a screen file for reading. If file was not found (?FILE returns false), a blank screen is created.

?LOADING SYS
Issue an error message if not loading.

?PAIRS SYS n1 n2 ---
Issue an error message if n1 does not equal n2. The message indicates that compiled conditional control directives do not match.

?STACK SYS
Issue an error message if the stack is out of bounds.

?TERMINAL --- f
Tests to see if the C64 STOP key is depressed. Returns true flag if it is; otherwise returns false flag.

@ 79 addr --- n "fetch"
Leave the 16-bit contents found at specified address.

ABORT SYS 79
Clear the stacks and enter the execution state. Return control to the operator's terminal, printing a restart message.

ABORTIO C64 SYS
Close all open I/O files. Used in event of an error.

ABS 79 n --- u "absolute"
Leave the absolute value of n as u.

AGAIN addr n --- (when compiling)
Used in a colon definition in the form:
: <defname> . . BEGIN . . AGAIN . . ;
At run-time, AGAIN forces execution to return to corresponding BEGIN . There is no effect on the stack at run-time. Execution cannot leave this loop.

ALLOT 79 n ---
Add the signed number to the dictionary pointer DP . N is the number of bytes to offset the pointer to reserve space in the dictionary or to re-origin memory.

AND 79 n1 n2 ---
Leave the bitwise logical AND of n1 and n2 as n3.

B/BUF SYS --- n
This constant leaves the number of bytes (1024.) per disk buffer, which is the byte count read from the disk by BLOCK.

B/SCR SYS --- n
This constant leaves the number of blocks per editing screen (1). By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters. C64-FORTH however edits and displays screens as 25 lines of 40 characters and 1 line of 24 characters.

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

BASE 79 --- addr
A user variable containing the current number base used for input and output conversion.

BASIC C64
Exit to BASIC. Interrupts are pointed back to ROM routines, and BASIC ROMs are re-enabled.

BEGIN 79 --- addr n (when compiling)
Used in a colon-definition in the form:

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

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL , AGAIN , or REPEAT . When executing UNTIL , a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs. There is no effect on the stack at run-time.

BKGRND C64 --- addr
Variable that contains the color value to which the C64 background color is set upon entering FORTH or executing COLD .

BL --- b "b l"
A constant that leaves the ascii value for a "blank".

BLANKS SYS addr count ---
Fill an area of memory beginning at addr with blanks.

BLK SYS 79 --- addr "b l k"
A user variable containing the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK 79 n --- addr
Leave the memory address of the block buffer containing block n. If the block is not already in memory, it is transferred from disk to which ever buffer was least recently written. If the block occupying that buffer has been marked as updated, it is rewritten to disk before block n is read into the buffer. See also BUFFER R/W UPDATE FLUSH

BLOCK-READ
BLOCK-WRITE SYS addr ---
Never used by the programmer.
These are internal routines used by (R/W) to read in or write out to disk or tape one 1024 byte block of data.

BORDER C64 --- addr
Variable that contains the color value to which the border on the C64 screen is set upon entry into FORTH or execution of COLD .

BRANCH SYS
Never used by the programmer.
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 , and REPEAT .

BUFFER 79 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 disk. BLOCK N IS NOT READ FROM DISK. BUFFER only reserves space for the block. The address left on the stack is the first cell within the buffer for data storage.

BUFFERS C64 SYS n ---
Changes the number of block buffers that memory is allocated for. n must be >= 2, and the maximum n is governed by how much free memory there is. Each buffer takes 1028 bytes. Updated blocks are written out to disk before the number of buffers is changed. C64-FORTH is shipped configured for 4 buffers.

C! 79 b addr --- "c store"
Store the lower 8-bits of b at address.

C, SYS b ---
Store lower 8 bits of b into the next available dictionary byte, advancing the dictionary pointer. This word is only available in FORTH systems implemented on byte-addressing machines.

C/L --- n
Constant that leaves the number of characters per line that the terminal is assumed to display.

C@ 79 addr --- b "c fetch"
Leave the 8-bit contents of memory address.

CALL C64 addr ---
Used to call a machine language routine from high level FORTH. addr may be a user generated subroutine or a subroutine in the I/O ROM space. The called subroutine must end with a RTS instruction. See CALLR

CALLRC C64 y1 x1 a1 addr --- y2 x2 a2
Used to call a machine language subroutine, as in CALL , however CALLR allows setting the Y, X, and A 6502 registers PLUS CLEAR THE CARRY BIT prior to calling the routine. The status of the three registers upon return from the subroutine are left on the stack. The called subroutine must end with a RTS instruction.

CALLRS C64 y1 x1 a1 addr --- y2 x2 a2
Sames as CALLRC except the CARRY BIT IS SET before the subroutine is called.

CBMEXPECT C64 addr count ---
Similiar to EXPECT except allows use of the normal Commodore 64 line editing capabilities. The line is not input until the RETURN key is pressed. For example, HERE 9 CBMEXPECT will return the first 9 characters found on the current display line when the RETURN key is pressed.

CFA SYS pfa --- cfa
Convert the parameter field address of a definition to its code field address.

CHRCLR C64 --- addr
Variable that contains the color code to which the C64 cursor color is set upon entry into FORTH or execution of COLD.

CKTOP C64 SYS --- f
Called by (R/W). If system device is a cassette, then inquires the operator whether should try searching for a screen data file on tape or just fill buffer in memory with spaces. This saves going through a whole cassette to learn that a screen has not yet been saved on it. f = 1 if user responded to look for file.

CKRDSTAT C64 --- f
Convenience word for checking disk status after a read. If disk status indicates an error, the status is printed, ABORTIO is executed, then ERROR is executed. If no error occured, then flag f returns 0 if o.k., or = 1 if end-of-file was encountered. Usefull after a INPUT# or GET# is executed.

CKST C64 --- f
Convenience word for checking the I/O status variable ST after an I/O transfer. If any status error other than end-of-file indication, an error message is printed; otherwise returns f = false if o.k., = true if end-of-file.

CKWRSTAT C64 ---
Similar to CKRDSTAT except no flag is returned since end-of-file condition does not occur when writing to a device. Useful after a PRINT# is executed.

CLOSCHN C64 SYS
Convenience word. If the SYSDEV# indicates I/O device is a disk, then CLOSCHN closes the error channel to the disk. Used in conjunction with OPENCHN .

CLOSE C64 file# ---
Functions same as CLOSE in BASIC. Closes a previously opened file referenced by file# .

CLRBUF C64 SYS addr ---
Fills a 1K block of memory with spaces. Called by (R/W) to clear a screen buffer when file is not found on disk during a read.

CMD# C64 file# ---
Similar to CMD# in BASIC, except that the output is directed to both the specified device and the terminal. This allows another output channel to be open while terminal output goes to a printer.

CMOVE 79 addr1 addr2 count ---
Move the specified quantity of bytes beginning at address from addr1 to addr2. The contents of addr1 is moved first, then proceeding toward high memory.

CODE C64
Used to create a FORTH callable definition composed of machine code. See ASSEMBLER glossary.

COLD SYS
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 SYS 79
When the word containing COMPILE executes, the execution address of the word following COMPILE is copied (compiled) into the dictionary. This allows special situations to be handled while compiling.

CONSTANT 79 n ---
A defining word used to create a data type that functions as a constant. Used in the following way:

```
n CONSTANT <constname>
```

<constname> is created with its parameter field containing n. When <constname> is later executed, the value of n will be pushed onto the stack.

CONTEXT SYS 79 --- addr
A user variable containing a pointer to the vocabulary within which dictionary seraches will first begin.

CONVERT SYS 79 d1 addr1 --- d2 addr2
Convert to the equivalent stack number the 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 non-convertible character.

COPY C64 n1 n2 ---
Copy the contents of screen n1 to screen n2 and mark n2 as updated.

COUNT 79 addr1 --- addr2 n
Leave the byte address addr2 and byte count n of a text string beginning at addr1. It is presumed that the first byte at addr1 contains the text byte count and the actual text starts at addr1+1. Typically COUNT is followed by TYPE.

CR 79
Transmit a carriage return to the selected output device.

CREATE 79
A defining word used in the form:

```
CREATE <defname>
```

by such words as CODE and CONSTANT to create a dictionary header for a new FORTH definition. The code field contains the address of its parameter field. The new word is created in the CURRENT vocabulary. Sometimes used with DOES> to create a new defining word:

```
: <typename> CREATE . . . DOES> . . . ;
```

CSP SYS --- addr
A user variable temporarily storing the stack pointer position, for compilation error checking.

CURRENT 79 --- addr
Leave the address of a user variable specifying the vocabulary into which new word definitions are to be entered.

D+ 79 d1 d2 --- dsum
Leave the double number sum of two double numbers.

D+- SYS d1 n --- d2
Apply the sign of n to the double number d1, leaving it as d2.

D- d1 d2 --- d3
Subtract d2 from d1 and leave the result as d3.

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

D.R d n ---
Print a signed double number d right justified in a field n characters wide.

D0= d --- f
Leave a true flag if double number d is equal to zero; otherwise leave a false flag.

D< 79 d1 d2 --- f
Leave a true flag if d1 is less than d2; otherwise leave a false flag.

D= d1 d2 --- f
Leave a true flag if d1 = d2; otherwise leave a false flag.

D> d1 d2 --- f
Leave a true flag if d1 is greater than d2; otherwise leave a false flag.

DABS d --- ud
Leave the absolute value ud of a double number.

DC C64
Stands for "disk command". Used as follows:

```
DC  "r0:prognew=progold"
```

Sends the string between the quote delimiters to the disk.

DECIMAL 79
Set the numeric conversion BASE for decimal input and output.

DEFINITIONS 79
Used in the form:

```
<vocname>  DEFINITIONS
```

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

DEPTH 79 --- n

Leave the number of the quantity of 16-bit values contained in the data stack, before n was added.

DIGIT SYS b n1 --- n2 tf (if ok)
b n1 --- ff (if bad)

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

DIR C64

Displays the directory of drive 0 of the device specified by SYSDEV# . If a dual drive is the configured on the system, the directory of that disk may be obtained by: DC "$1"

DKFLAG C64 --- addr

Variable used as a flag for disk type. DKFLAG is normally =0 for a 1541 type of disk. To use a 1540 1540 type disk without an upgrade ROM, set = 1.

DLITERAL SYS d --- d (when executing)
d --- (when compiling)

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

DMAX d1 d2 --- dmax "d max"

Leave the larger of two double numbers.

DMIN d1 d2 --- dmin "d min"

Leave the smaller of two double numbers.

DNEGATE 79 d1 --- d2

Convert d1 to its double number 2's complement.

DO 79 n1 n2 --- (when executing)
--- addr checkdigit (when compiling)

Used within a colon definition as follows:

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

DO is used to mark the loop back point of a program loop. When compiling, DO compiles the procedure (DO) into the dictionary and pushes the current dictionary address and a check digit onto the stack. When executing, (DO) takes the limit n1 and the starting loop index n2 off the data stack, pushes them onto the return stack. The FORTH words up the point marked by LOOP, /LOOP, or +LOOP are executed. Upon reaching loop point, the index is incremented by one in the case of LOOP , or n is added to the index in the case of +LOOP or /LOOP. Until the new index equals or exceeds the limit, execution loops back to just after the DO point; 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, executing 'I' will push the current loop index onto the data stack. 'J' and 'K' will push the indexes of the second and third outer loops.
See I J K LOOP +LOOP /LOOP LEAVE

DOES> 79 "does"
Used with CREATE to create a new word type:

```
: <typename>  CREATE  . . .
              DOES>   . . . ;
```

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

```
<typename>  <defname>
```

uses CREATE to create a dictionary entry for <defname>. The FORTH definitions between CREATE and DOES> are executed at this point, such as to reserve memory for a data array, etc. When <defname> is later executed, its parameter field address is pushed onto the stack, and the FORTH definitions specified between the DOES> and ; portion of <typename> will be executed.

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

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

DR0
Sets disk drive OFFSET to zero. Actually the I/O routines in C64-FORTH don't use the value in OFFSET. However, for compatibility OFFSET must be set to zero, so DR0 exists in the dictionary to do this.

DROP 79 n ---
Drop (remove and discard) the top value on the stack.

DU< ud1 ud2 --- f "d u less"
Flag is left true if ud1 is less than ud2. Both numbers are unsigned.

DUMP addr count ---
Print the contents of count number of memory locations beginning at addr. Both addresses and contents are shown in HEX to allow reasonable formatting on the C64 screen.

DUP 79 n1 --- n1 n1
Duplicate the top value on the stack.

DV1
DV2
DV3
DV4 C64 --- addr
Variables used to hold I/O unit characteristics. C64-FORTH may have up to 4 I/O devices attached for saving screens. These are actually double variables holding 4 bytes of information. The addr returned when one of these variables is executed points to a 16-bit value indicating how may screens this device is to hold. addr+2 contains the device # of the unit. addr+3 contains the drive #, allowing access to the second drive of a dual drive unit if attached. See TECHNICAL INFORMATION section of manual.

EBUF C64 --- addr
A buffer of 30 bytes in length which various I/O routines in C64-FORTH use for saving disk status, etc.

EDIT C64 scr# ---
Used to enter editing mode on a screen. See manual section on EDITOR .

ELSE 79 addr1 n1 --- addr2 n2 (when compiling)
Used within a colon-definition as follows:

IF . . . ELSE . . . ENDIF

ELSE marks the end of the FORTH words that are executed when the IF condition tests true, and the start of the FORTH words that are executed when the IF condition tests false. There is no effect on the stack at run-time.

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 br calculating the offset from addr1 to HERE and storing at addr1.

EMIT 79 b ---
Transmit ascii character b to the selected output device. OUT is incremented for each character output.

EMPTY-BUFFERS 79
Marks all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the disk.

EMSGS C64 --- addr
Searched for by MESSAGE . Contains ascii error messages. May be FORGOTten if neccessary. MESSAGE automatically accomodates its absence. See MESSAGE .

ENCLOSE SYS addr1 b --- addr1 n1 n2 n3

The text scanning primitive used by WORD . From the text address addr1 and an ascii delimiting character b, is determined the byte offset n1 to the first non-delimiting character, the offset n2 to the first delimiter after the text string, and the offset n3 to the first non-delimiter after that. This proceedure will not process past an ascii 'null' treating it as an unconditional delimiter.

END

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

ENDIF addr n --- (when compiling)

Used in a colon-definition as follows:

```
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 .

At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at addr. N is placed on the stack for error checking.

ERASE addr count ---

Clear (fill with null bytes) a region of memory starting from addr for count number of bytes.

ERROR SYS line# --- in blk

Execute error notification and restart of system. WARNING is first examined. If WARNING = -1, the definition (ABORT) is executed, which executes the system word ABORT . The user may cautiously modify this execution by altering (ABORT) .

If WARNING didn't equal -1, (it is normally set to 1), then MESSAGE is executed. However, the function of MESSAGE will be described here:

If WARNING = 0, line# is just printed as a message number. If WARNING = 1, then first MESSAGE checks to see if a definition by the name of EMSGS exits in the dictionary. EMSGS is comprised of ascii strings, each one being preceeded by a number. The strings are scanned searching for a number matching line#. If one is found, the that message is printed as the error. If either EMSGS wasn't found, or a matching error number wasn't found, then screen #4 is read in from disk or tape (or screen #5 if the line# is > 24). The 40 character line referenced by line# (or line#-25 if >24) is output as the error message.

Final action of ERROR is the execution of QUIT .

EXECUTE SYS 79 addr ---
Execute the definition whose code field address is on the stack. The code field address is also called the compilation address.

EXIT 79
Used within a colon definition that at run-time will terminate execution of that definition at that point. MAY NOT be used within a DO loop.

EXPECT 79 addr count ---
Transfer characters from the terminal to address addr, until a "return" or the count number of characters have been received. One or more nulls are added at the end of the text.

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

FILL 79 addr count b ---
Fill memory at the address with the byte value b for count number of bytes.

FIND 79 --- addr
Leave the compilation address (CFA) of the next word name found in the input stream. If that word cannot be found in the dictionary after a search of CONTEXT then FORTH then CURRENT vocabularies, a zero is returned.

FIRST SYS --- n
A constant that leaves the address of the first (lowest) block buffer.

FLD SYS --- addr
A user variable for control of number output field width.

FORGET 79
Used as follows:

```
FORGET <defname>
```

Deletes definition named <defname> from the dictionary with ALL entries physically following it (defined after it in ANY vocabulary). See FENCE

FORTH 79
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.

GET C64 --- b
Reads keyboard and returns character pressed. Does not wait for a key to be pressed, so returns a null value if none is.

GET# C64 file# addr count ---
Used to input bytes from an external file. count specifies a maximum number of bytes to input. Input terminates only upon reaching that count or upon error. The received bytes are stored at addr+1, with the count of the number of bytes received left at addr.

GETDEV C64 file# --- dev#
Used by (R/W) to get the device # that the specified file# was opened with.

HERE 79 --- addr
Leave the address of the next available dictionary location.

HEX
Set the numeric conversion base to sixteen (hexadecimal).

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

HOLD 79 b ---
Used between <# and #> to insert an ascii character onto the text string being created during numeric output conversion. e.g. 2E HOLD will place a decimal point character onto the string.

I 79 --- n
Used within a DO-LOOP to place the current loop index of the innermost DO loop onto the data stack.

ID. SYS addr ---
Print a definition's name from its name field address.

IF 79 f --- (at run-time)
--- addr n (when compiling)
Used in a colon-definition as follows:

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

At run-time, IF selects execution based on a flag on the stack. If f is true (non-zero), execution continues ahead thru the true part (tp). If f is false (zero), execution skips till just after the ELSE (if one was specified) to the false part (fp), or to just after the ENDIF part. In either case, execution continues just after the ENDIF part.

IMMEDIATE SYS 79
Mark the most recently made definition so that when encountered at compile time, it will be executed, rather than being compiled into the definition. The user may force compilation of a definition marked as IMMEDIATE by preceeding it with [COMPILE] .

INDEX scr1 scr2 ---
Print the first line of each screen in the range starting with scr1 through scr2. Traditionally, the first line of a text screen is used for comment on the function of the code in that screen.

INPUT# C64 file# addr count ---
Transfers count maximum number of characters from file# to addr+1. Input is terminated upon reading return or null, reaching count, or error. Count of actual number of bytes received is left at addr. If a null or return terminated input, it is left at the end of the string in memory, but is not included in the count of bytes received.

INTERPRET SYS
The outer text interpreter which sequentially executes or compiles text from the input stream (from terminal or disk) depending on STATE. If the word name cannot be found after a search of the CONTEXT and then CURRENT vocabularies, it is converted to a number according to the current base. That also failing, an error message 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 on the stack. The decimal point has no other purpose than to force this action.

J 79 --- n
Return the index of the next outer DO loop. May only be used within a nested DO-LOOP in the form:
DO . . . DO . . . J . . . LOOP . . . LOOP

K C64 --- n
Return the index of the second outer DO loop. May only be used within a nested DO-LOOP in the form:
DO . . DO . . DO . . K . . LOOP . . LOOP . . LOOP

KEY 79 --- b
Leave the ascii value of the next key struck on the terminal keyboard.

KEYIN C64 --- b
Waits for a key to be depresses on the keyboard. The cursor is turned on while waiting. As soon as a key is depressed, the cursor is turned off and the routine returns. Gives visual indication that the program is waiting for one key to be pressed.

KEYRANGE C64 --- b
Routine called internally by EXPECT to wait for a key to be pressed, and return with its value only if it was an ASCII character, the DEL key, or RETURN.

LATEST SYS --- addr
Leave the name field address of the topmost word in the CURRENT vocabulary.

LEAVE 79
Force termination of a DO-LOOP at the next oppertunity by setting the loop limit equal to the current value of the loop index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA SYS pfa --- lfa
Convert the parameter field address of a dictionary definition to its link field address.

LFFLG C64 --- addr
Variable used as flag that controls whether a LINE-FEED character is sent to the current CMD# device following any RETURN character. LFFLG = 0 disables line-feed, <> 0 enables line-feed.

LIMIT SYS --- n
A constant that leaves the address just above the highest memory address used for the disk buffers. This is usually the start of the user variables area.

LIST 79 n ---
Display the ascii text of screen n on the terminal. LIST sets the variable SCR to the screen number requested.

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

LITERAL SYS 79 n --- (when compiling)
If compiling, then compile the stack value n as a 16-bit literal. This definition is immediate so that it will execute during a colon definition.

LOAD 79 n ---
Read in screen n, and treat it as if it had been typed in at the terminal. LOAD is used to initiate compiling of source code on screens. Loading of a screen will terminate at the end of the screen or at ;S .
See ;S and -->

LOOP 79 addr n --- (when compiling)
Used in a colon definition as follows:

```
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) into the dictionary and uses addr to calculate an offset to DO. n is used for error checking.

M* n1 n2 --- d
A mixed magnitude math operation which leaves the double number product of two signed numbers.

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

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

MAX 79 n1 n2 --- max
Leave the greater of two numbers.

MEMTOP C64 SYS addr ---
Changes the top of memory allowed for the FORTH system. Usual top for FORTH is D000, which is the highest allowable RAM address. If desired, the top may be lowered to reserve a section of RAM that the be out of range of the FORTH system. addr must not push the FORTH system too low. Above HERE there must be room left for the text output buffer starting at PAD, the block buffers, each of which consumes 1K, and the user area (128 bytes long). MEMTOP executes RESTART, then COLD to make the new value permanent.

MESSAGE n ---
The operation of MESSAGE is fully described under ERROR.

MIN n1 n2 --- min
Leave the smaller of two numbers.

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

MOVE 79 addr1 addr2 count ---
Move the contents of count number of 16-bit memory locations from addr1 to addr2. The contents of addr1 is moved first. count must be < 16K words (32K bytes).

NEGATE 79 n --- -n
Leave the two's complement of a number.

NFA SYS pfa --- nfa
Convert the parameter field address of a definition to its name field address.

NOT 79 f --- f
Reverse the boolean value of f. This is identical to 0= .

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

OFFSET --- addr
A user variable which may contain a block offset to disc drives. C64-FORTH does not use its contents to determine which I/O unit a certain screen resides on, however it must be kept = 0 since BLOCK adds it to the requested block number.

OPEN C64 file# dev# sa addr count ---
Open a file to an external device. dev# is the device number that the file # will be associated with, sa is the secondary address that will be sent to the unit. addr and count specify the location and length of a filename string to be sent to the device. If no filename is to be sent, count should equal zero.

OPENCHN C64
Convenience word. Opens an error channel to the device in SYSDEV# only if it is a disk (i.e. SYSDEV# > 3). Used with CLOSCHN .

OR 79 n1 n2 --- or
Leave the bit-wise logical or of two 16-bit stack values.

OUT --- addr
A user variable that contains a value incremented every time EMIT is called. The user may initialize and examine OUT to control display formatting.

OVER 79 n1 n2 --- n1 n2 n1
Copy the second stack value, placing it as the new top.

PAD 79 --- addr
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.

PICK 79 n1 --- n2
Return the contents of the n1-th stack value, not counting n1 itself. e.g. 2 PICK is eqivalent to OVER .

PREV SYS --- addr
A variable containing the address of the disk buffer most recently referenced. The UPDATE command marks this buffer to be later written to disk.

PRINT# C64 file# addr count ---
Output count number bytes to the file, starting from addr.

QUERY 79
Input 80 characters of text (or until a "return") from the terminal. Text is positioned at the address contained in TIB with IN set to zero.

QUIT 79
Clear the return stack, stop compilation, and return control to the operators terminal. No message is given..

R# --- addr
A user variable which may be used to hold the position of the editing cursor, if a different editor is used, or may be used for other file related functions.

R/W SYS addr blk f ---
Used to read in or write out one screen block. addr specifies the starting address of the block, blk is the block number, and f is a flag that is set =0 to write the block out to disk, or =1 to read in the block. R/W automatically generates the proper filename for reading or writing the block to disk or tape, transfers the data file, and does error checking.

R0 --- addr "r zero"
A user variable containing the initial location of the return stack. Used for clearing the return stack.

R> SYS 79 --- n "r from"
Remove the top value from the return stack and leave it on the data stack. See >R and R

R@ 79 --- n
Copy the top of the return stack to the data stack.

RDROP C64 SYS
Removes and discards the top value on the return stack. Care must be used in the use of RDROP .

RDSTAT C64 file# ---
If system device is a disk (i.e., SYSDEV# > 3), reads in the disk status and places it in EBUF .

REBOOT C64
Updates the permanent boot-up values that specify the system characteristics. All new definitions defined up to this point will be made a permanent part of the system in memory. Called by SAVESYSTEM .

REPEAT 79 addr n --- (when compiling)
Used within a colon definition as follows:

```
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 checking.

ROLL 79 n ---
Rotate the top n stack items. e.g. 3 ROLL is the same as ROT .

ROT 79 n1 n2 n3 --- n2 n3 n1 "rote"
Rotate the top three values on the stack, bringing the third to the top.

RP! SYS
A procedure that initializes the return stack pointer from the value in user variable R0 .

S->D n --- d
Sign-extend a single number to form a double number.

S0 SYS --- addr "s zero"
A user variable that contains the initial value for the data satck pointer.

SAVE-BUFFERS 79
Write all blocks to disk that have been marked as updated.

SAVESYSTEM C64
Used to save the current FORTH system onto disk or tape as follows:

```
SAVESYSTEM "filename"
```

All recently defined definitions are made a permanent part of the FORTH system.

SAVETURNKEY C64

Used to save an application program onto disk or tape as follows:

```
SAVETURNKEY "filename"
```

The program saved is not a usable FORTH system. The last word defined in the vocabulary is executed immediately upon booting up. Executing SAVETURNKEY destroys the current FORTH system, so when done saving onto disks, the FORTH system will have to be rebooted.

SCR 79 --- addr

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

SETDEV# C64 blk#1 --- blk#2

Translates blk#1 into blk#2 by doing a check of variables DV1, DV2, DV3, and DV4. See technical information section of manual.

SIGN 79 n d --- d

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 SYS

Used during word definition to toggle the "smudge bit" in a definitions' name field. This prevents an uncompleted definition from being executed, however the name will show up during a VLIST. If the most recently created definition was not completed, and cannot be FORGOTten, then execute SMUDGE. The header of the unfinished definition can now be removed via FORGET .

SP! SYS

A procedure to initialize the data stack pointer from S0 .

SP@ --- addr

A procedure to return the data stack pointer (position of the current data stack before the addr was added). Used in the following way:

```
SP@ .
```

is useful in debugging a FORTH routine involving DO loops to see if stack is getting "over-filled" or "over-empty" from unbalanced use of stack values.

SPACE 79

Output an ascii blank character to the terminal.

SPACES 79 n ---

Output n ascii blanks to the terminal.

ST C64 --- addr
A variable into which the C64-FORTH I/O routines place the I/O status byte after each operation. This variable may be checked after an I/O operation the same way it is done in BASIC .

STATE SYS 79 --- addr
A user variable containing the compilation state. A non-zero value indicates in compilation mode.

SWAP 79 n1 n2 --- n2 n1
Exchange the top two stack values.

SYSDEV# C64 --- addr
A variable that is set to the currently active system I/O device. The reading or writing of blocks will automatically set this variable. However, any user use of data files or executing the SAVESYSTEM or SAVETURNKEY words must have SYSDEV# preset to the proper I/O device number.

TAPEFLG C64 --- addr
A variable used only for cassette I/O. If set =0, then any block I/O automatically goes to the cassette. If <>0, CKTOP will, if a block is to be read in, ask the user if should look for the block on tape or create a new blank one. This allows creating new screens that don't already exist on tape.

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 79
See ENDIF .

TIB SYS --- addr
A user variable containing the address of the terminal input buffer.

TOGGLE addr b ---
Complements the contents of addr by the bit pattern b. Effectively an exclusive-or of b with the contents of the location.

TON C64 ---
Sets a flag so that FORTH TRACE mode turns on at the start of the execution of the next word in the input stream that is defined as a colon definition.

TRAVERSE SYS addr1 n --- addr2

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

TYPE 79 addr count ---

Output count number of characters from memory starting from addr.

U* 79 u1 u2 --- ud

Leave the unsigned double number product of two unsigned numbers.

U. 79 u ---

Unsigned-number print.

U.R u n ---

Print unsigned-number right-justified in field of n characters.

U< 79 u1 u2 --- f

Unsigned less-than. f is true if u1 is less than u2. Works correctly even if numbers differ by more than 32K.

U/ ud u1 --- u2 u3

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

U/MOD 79 ud u1 --- u2 u3

Identical to U/ .

UNTIL 79 f --- (run-time)
addr n --- (when compiling)

Used within a colon definition as follows:

```
BEGIN . . . UNTIL
```

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

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

UPDATE 79

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

UPPER C64 addr count ---
Scans through byte string starting at addr for count number of characters and converts any shifted (from entering characters with the C64 SHIFT key held down) characters to unshifted ones. Makes string matching more user-forgiving.

USE SYS --- addr
A variable containing the address of the block buffer to use next, i.e. the one least recently used.

USER n ---
A defining word used to create a new user variable as follows:

```
n USER <varname>
```

The parameter field of <varname> contains n as a fixed offset relative to the start of the user area. When <varname> 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 79 --- addr
A defining word used to create a variable as follows:

```
VARIABLE <varname>
```

When VARIABLE is executed, it creates the definition <varname> with its parameter field not initialized to to any particular value. When <varname> is later executed, the address of its parameter field is left on the stack so that a fetch or store may access this location.

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

VOCABULARY 79
A defining word used as follows:

```
VOCABULARY <vocname> IMMEDIATE
```

A new vocabulary named <vocname> is created. When the vocabulary name is executed, it becomes the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "<vocname> DEFINITIONS" will also make <vocname> the CURRENT vocabulary into which new definitions are placed.

<vocname> will be chained as to include all definitions of the vocabulary in which <vocname> itself was defined. All vocabularys 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. The C64 CTRL key will slow the listing, and the STOP key will terminate listing.

WARM C64

Execute a WARM start - stacks are cleared and control returns to the FORTH input interpreter, but the disk buffers are NOT cleared.

WARNING --- addr

A user variable containing a value used to control error and informative message printing. See ERROR .

WHERE C64 n blk ---

If executed immediately after an error while loading in a screen, will show the line and location of the error.

WHILE 79 f --- (run-time)
addr1 n1 --- addr1 n1 addr2 n2 (when compiling)

Used in a colon definition as follows:

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

At run-time, WHILE selects conditional execution based on the 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 compiles OBRANCH and leaves addr2 of the reserved offset. The stack values will be resolved by REPEAT .

WIDTH SYS --- addr

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 thru 31, with a default of 31. The name character count and its natural characters are saved, up to the value in WIDTH . The value may be changed at anytime within the above limits.

WORD 79 b --- addr

Scan off the next text characters from the input input stream, until a delimiter b is found. The string is stored starting at HERE+1, with the byte count stored at HERE. The actual delimiter encountered (b or a null) is stored after the string, but is not included in the byte count. Leading occurances of the delimiter b are ignored. If BLK is zero, text is taken from the teminal input buffer, otherwise from the disk block stored in BLK . The addr returned is the address of HERE. See BLK IN

X "null"

This is a pseudonym for the "null" definition. When an ascii null (byte = 0) is encountered in the input stream while interpreting, this definition is executed. Interpretation of the text from the terminal input buffer or within a disk buffer is terminated, and FORTH goes into the wait-for-input mode.

XOR 79 n1 n2 --- xor

Leave the bit-wise logical exclusive-or of two values.

[SYS 79

Used within a colon definition in the form:

: <name> [words] rest of def ;

Temporarily turns off compile mode to allow using FORTH words for calculating an address, etc. to qualify the compilation when turned back on by] . The words between [and] are interpreted, not compiled.

[COMPILE] SYS 79

Used within a colon definition as follows:

: <name> [COMPILE] FORTH ;

[COMPILE] will force compilation of a definition marked as IMMEDIATE, that would otherwise execute during compilation. The above example will select the FORTH vocabulary when <name> executes, rather than at compile time.

] SYS 79

Resume compilation that was suspended by [. See [