.>>> S K - F O R T H 7 9 <<<<

USER'S GUIDE AND GLOSSARIES

RELEASE 2.0

FORTH-79 STANDARD

with

Double-Number Standard Extensions

BY:

JOHN W. BROWN

DISTRIBUTED BY:

SATURN SOFTWARE LIMITED
P. O. BOX 397
NEW WESTMINSTER, V3L 4Y7

DISCLAIMER

Saturn Software Limited and the author make no warranties, either expressed or implied, with respect to this manual, or with respect to the software it describes, or its quality, performance, or suitability for any particular application. In no event will Saturn Software Limited or author be liable for any direct, indirect, incidental, or consequential damages resulting from any defects in the manual or the software supplied.

COPYRIGHT NOTICE

Address all communications to:

John W. Brown, President
Saturn Software Limited
P. O. BOX 397
New Westminster, B. C.
V3L 4Y7, CANADA

INTRODUCTION

FORTH is a complete programming system. FORTH consists of an extensible programming language, an interactive operating system, a text editor, a built-in assembler, and, if a disk is present, virtual memory. FORTH is a consistent and complete programming environment which can be extended for each new application. FORTH turns your computer into a dual 16 bit stack computer. The user has immediate access to the parameter stack, which uses reverse Polish conventions for all operations. All programming in FORTH consists of defining new FORTH "words", which, when executed, produce the desired results. All new words defined by the user are equal, in all respects, to the words already existing in the FORTH vocabulary, and can, in turn, be used to define other FORTH words.

EQUIPMENT REQUIRED

SK-FORTH79 software requires the following minimum microcomputer hardware system to function:

1. SYM-1 single board computer.

2. A KTM-2 video terminal board (or equivalent) and suitable monitor. Purchaser assumes responsibility for adapting SK-FORTH79 to a different terminal.

3a. Cassette based mass storage systems require at least 24K of memory located at $0000 through $6000 and one (preferably two) remote controlled cassette recorders. The optional second recorder should be installed as described in the RAE reference manual.

3b. Disk based mass storage systems require at least 32K of memory located at $0000 through $8000 and the dual HDE mini disk system with the FODS operating system software loacated at $7000 through $8000.

4. (Optional) A printer which will operate on the existing SYM-1 20 ma current loop interface.

GETTING STARTED

1. Take the cassette containing the SK-FORTH79 object code and insert it in the cassette recorder making sure that it is fully rewound. There are three copies of the object code on the cassette. Now load the object code using the monitor .L2 command. The file identification number is 1 but this can be omitted. Users with disk systems should boot FODS and exit to the monitor before loading the object code.

2. The software assumes that your terminal is on the standard SYM-1 RS232 serial port and that the terminal is operating at 4800 baud. The only intelligent function you terminal must possess is the ability to move back one space on the screen when sent the backspace character. If your terminal operates at a different baud rate or is super dumb you should modify the following memory locations before

cold starting FORTH.

KTMFLG	$0240	1=Intelligent(KTM-2), 0=Dumb. In the dumb mode a back slash is echoed for the delete key, the backspace key is deactivated and the erase to end of line feature is deactivated.
TBAUD	$0241	Set to $01 for 4800 baud.
PBAUD	$0242	Set to $06 for 2400 baud. See the SYM-1 reference manual for other values. (ie $D5=110 baud, and $4C=300 baud)

3. Now cold start the SK-FORTH79 with the monitor command .G 200 (ret). Remember your system must have at least 24K to cold start SK-FORTH79. You should hear a beep and see the SK-FORTH79 copyright notice.

4. Now do step (a), (b) or (c) below depending on your system:

a) 24K-CASSETTE (ret)
   BOOT (ret)

b) 32K-CASSETTE (ret)
   BOOT (ret)

c) 32K-DISK (ret)
   BOOT (ret)
   (Remember, FODS software must be in memory!)

5. All users may now try some FORTH. You would be advised however to resave the customized object code to mass storage so that on subsequent loads it is only necesary to type the word BOOT. Do this as follows:

a) Cassette based systems; Place a blank rewound cassette in your recorder and advance past the tape leader and set in the record mode. Now type the following to save 3 copies of the system to tape with a file identification number $01.

```
1 SYSSAVE 1 SYSSAVE 1 SYSSAVE (ret)
```

b) Disk based systems; Place the diskette in drive 1 with FODS operating system software as the first file (mine is called %V3.1) and the utilities %DIR and %LDN. There should be no other files on this disk as FORTH uses the second half of this disk for 35 more screens. Now do the following to save FORTH to disk:

```
Exit FORTH with        MON (ret)
Enter FODS with        .G 7380 (ret)
Save FORTH with        ENT 1/$0200$3000=%SKF
Back to FORTH with     SKF (ret)
Initialize buffers     BOOT (ret)
```

At this time you should place a blank formated diskette in drive number 2. This diskette will contain screens 1 through 70. Note

that each screen must be CLEARed before it is used. See the section on the editor when you are ready to use screens.

Miscellaneous Notes

1. 24K cassette systems have 4 simulated disk in ram screens from $5000 through $6000.

2. 32K cassette systems have 8 simulated disk in ram screens from $6000 throught $8000.

3. 32K HDE-FODS disk system users have no simulated disk in ram screens.

4. Disk users may switch to cassette operation by setting the system variable DISK to 0. To go back to disk operation set it equal to 1. Using screens 5 6 7 or 8 while in cassette mode will wipe FODS and result in a system crash when you go back. Using cassette commands while set for disk operation will produce unpredictable results.

5. We highly recommend the book "Starting FORTH" by Leo Brodie, of FORTH, Inc. for its extremly readable introduction to FORTH. The book is published by Prentice Hall and should be available at your local computer store. There is also a bibliography of FORTH related material at the end of this manual.

LINE EDITING COMMANDS

SK-FORTH79 is equipped with an extremly versatile input line editor. Control codes used to direct the cursor to any point within an input line for insertions and/or deletions. All included features are documented below.

Control A

Move cursor to the start of the current input field and set the input buffer pointer to zero. This operation does not clear or cancel the current input line being prepared.

Control H (backspace)

Back up (tab back) one position on the screen and in the input buffer if possible. This command has no effect on the line buffer contents. It serves only to move the cursor back in a line for possible insertions and/or deletions. If the cursor is already at the start of a line the cursor will wrap around and appear at the end of the line.

Control L

Clear the console screen. This command has no effect on the current line being prepared. If executed during the preparation of an input line the screen will clear and the line being prepared will appear at the top of the screen awaiting completion.

Control I (tab)

Tab forward one positon in buffer and on the screen updating the current screen position if necessary. If you try to tab past the end of a line the cursor will wrap around to the beginning of the line. You cannot tab through an empty buffer. This command does not insert blanks, it simply serves to move the cursor forward through the line to a position where insertions/deletions are required.

Control M or RETURN

Send buffer contents to FORTH. Line is truncated at the current cursor position. If you wish to send the entire line the type Control Z before you hit RETURN.

Control S

Escape to the SYM-1 or KIM-1 Monitor. To return to FORTH type .G 0 (ret) on the SYM. For the KIM push the space bar and type G . Return is through FORTH's warm entry point and the system not be configured the same as when you left (in particular the system base will be 10 and FORTH will be the context vocabulary). The FORTH word MON will exit and return you to the system in the same state as when you left.

Control X

Cancel current input line, clear buffer, and position cursor at the start of the input field. Control X does not return control to FORTH, only to the start of the input line routine. To cancel a line and return to FORTH type Control X and then RETURN.

Control Z

Move cursor from its current position to the end of the current input line updating the screen if necessary then erase to the end of the input field and position the cursor at the end of the input line. Use this command after any deletions or insertions to clean up the display.

ESC (the escape key)

ESC followed by any other input character: Skip through the buffer from where you are, updating the screen as you go, and stop just after the first occurance of the character which followed ESC. Use this command after Control A for rapid access to a point where an insertion or deletion is required.

RUBOUT or DELETE

Delete the character behind the cursor off the screen and out of the input buffer and close up the space created in the buffer. Note that the space on the screen is not closed. If this is desired Control S after deletions.

BREAK

Used to halt an executing FORTH program that as included the word ?TERMINAL or to halt the trace feature. After hitting BREAK push the RETURN key to abort or push the SPACE bar to continue execution.

All other control characters and function keys are trapped and ignored. All non control characters are inserted in the input buffer and echoed to the terminal screen.

The default option of the line editor is character insertion. This means that if you backspace or otherwise move to a point within an input line, any characters typed will automatically inserted at that point in the input buffer, even though on the screen it appears that you are over-typing existing characters!! To observe that the characters are indeed inserted type .

Characters are fetched from the console with the echo turned off. Only after the character has been analyzed, and found to be valid, will it be echoed back to the console CRT.

STACK MANIPULATION

The best way to think of the FORTH parameter stack (also referred to as the number stack or the data stack) is to compare it with the stack on a Hewlett Packard Reverse Polish calculator. The major difference is that in SK-FORTH79 the parameter stack will hold about 63 16-bit integers. To enter numbers on the stack, simply type them at the terminal, separated by spaces. Hit return (ret), and they are entered.
Example: Type the following:

```
1 4 5 10 (ret) OK
```

Note that (ret) means for you to push the return key and does not, in fact, appear on the screen. After (ret), FORTH replies with OK. Actually, OK would appear one space after the 10, but it will always be shown as above, so that you can tell when the return key was pushed. Note also that OK is typed by the FORTH system to indicate that it has processed your command. OK is never typed by the operator. In the example above and the ones which follow the characters typed by the computer are underlined so that you can tell who is doing what.

The last number entered is on the top of the stack (10 in this case). 5 is at top-1, 4 at top-2, and 1 is at the bottom of the stack. To get the numbers back, use the print number command (.). The print number command is a "period". Many FORTH words (commands) are punctuation marks. In fact, a FORTH word can be made up of any symbols you desire. In order to distinguish between regular punctuation and a FORTH word which looks like a punctuation mark, regular rounded brackets will be used. The only current use that SK-FORTH79 makes of rounded brackets is for comments. Again, the FORTH print number command is a single period, which we denote by (.).
Example: Type the following:

```
. . . . (ret) 0 5 4 1 OK
```

The top of the stack is always the first out and the bottom is last.

Now try the following stack operations and verify that they work as described. Pay careful attention to the notation which will be used throughout this manual and the glossaries.

SWAP Reverse top two stack items.

```
top-1  top      word              top-1  top
 n1     n2      SWAP      ->      n2     n1
stack before  execute   results  stack after
(or inputs)                     (or outputs)
```

In the notation above, which will be used throughout this manual, the top of the stack will always be on the right hand side of the designated stack inputs and outputs. n1 and n2 above represent 16 bit signed integer numbers. At the terminal you would observe the following:

```
4 8 (ret) OK
SWAP (ret) OK
. . (ret) 4 8 OK
```

No, there is not a mistake; the above is exactly what you would see. Remember, the top of the stack is always the first out!! If the above is performed without the SWAP, the following would be be produced:

```
4 8 . . (ret) 8 4 OK
```

The remaining parameter stack operations will now be presented in ever decreasing detail. Try each example at your terminal and make up more of your own. A complete understanding of the operation of the parameter stack is necessary before attempting any serious FORTH programming.

DUP Duplicate top of stack.

```
 top-1  top                  top-2   top-1   top
  n1     n2       DUP  ->     n1      n2     n2
 stack inputs                stack outputs
```

Verify the following at your terminal:

12 25 DUP . . . (ret) 25 25 12 OK

Remember in the notation above that the top of the stack is always to the right of both the stack inputs and outputs. When you try the examples at the terminal the print number command (.) will always print the top of stack first, hence the apparent reversal.

DROP Throw away top of stack.

```
  top-1    top                  top-1    top
   n1       n2      DROP  ->     ??       n1
```

Try the following:

11 22 DROP . . (ret) 11 0 OK

Note: The 0 indicates that the bottom of the stack has been reached. One more print number command will produce an error!! Either put the zero back again or go ahead with the print number command and the error recovery routine will restore the zero to indicate the stack bottom.

OVER Make copy of second item on top.

```
  n1    n2    n3     OVER  ->    n1    n2    n3    n2
              top                                  top
```

At the terminal you would observe the following.

11 22 33 OVER . . . . (ret) 22 33 22 11 OK

ROT Rotate stack. top-2 ->top; top ->top-1; top-1 ->top-2

```
  n1   n2   n3    ROT   ->    n2   n3   n1
             top                      top
```

Here is an example to try.

123 456 789 ROT . . . (ret) 123 789 456 OK

Did I hear you say they didn't rotate? Remember, the top comes out first with the print number command, so 123, which was third from the top, did indeed get rotated to the top. Check the others and then

make some examples of your own. When you write a FORTH program you will be using lots of SWAP's, ROT's, DROP's, and OVER's.

?DUP Duplicate top of stack only if it is non zero

```
  n1    n2    ?DUP  ->    n1    n2    ??
```

Try these:

12 13 ?DUP . . . (ret) 13 13 12 OK

12 0 ?DUP . . . (ret) 0 12 0 OK

In the first case the top of the stack is duplicated because it is non zero. In the second there is no change in the stack. Note that the last zero in the second example indicates the bottom of the stack. Try one more print number command and you will get an error.

THE RETURN STACK

The return stack is used mainly by the FORTH system and advanced FORTH programmers. It is used for the nesting of colon definitions and DO loops (more on these later), among other things. It can be used for temporary storage within a single routine provided that it is restored to its previous state before leaving. Listed below are the operations which involve the return stack. You should not find that you need to use the return stack for elementary programming. Warning, using >R and R> directly from the console there must be a balanced number or >R's and R>'s before RETURN is pushed of the FORTH system will crash! The same is true within each colon definition and DO...LOOP.

>R Move top of parameter stack to the return stack.

```
   n1    >R   ->   ---
```

The dashes indicate that the top of the parameter stack has been removed. R> Retrieve item from return stack.

```
   ---    R>  ->   n
```

R Copy top of return stack to parameter stack (state of return stack is unchanged).

```
   ---    R   ->   n
```

BASIC ARITHMETIC OPERATIONS

Basic operators for add, subtract, multiply, and divide, are +, -, *,

and /, respectively. Below are their definitions in terms of input and output stacks and some examples for you to verify.

```
+       Add top of stack to top-1 (top and top-1 lost)

    n1    n2    +   ->   sum

-       Subtract top of stack from top-1 (top and top-1 lost)

    n1    n2    -   ->   difference

*       Multiply top of stack by top-1 (top and top-1 lost)

    n1    n2    *   ->   product

/       Divide top into top-1, integer division, remainder lost
        (top and top-1 lost)

    n1    n2    /   ->   quot     (quot=n1/n2)
```

Verify the following examples:

```
12 23 + .  (ret) 35 OK
44 11 - .  (ret) 33 OK
10 12 * .  (ret) 120 OK
45 13 / .  (ret) 3 OK    (remember, integer division)
```

By now you should thoroughly understand the structure of the FORTH parameter stack. You should understand the concepts of stack inputs and stack outputs and the notation used in explaining the operation of FORTH words (i.e., functions, operators, etc.). You should be able to study the glossaries, pick out words, and verify their functions (well, some of the easier to understand ones, anyway).

NON DESTRUCTIVE STACK PRINT

You probably wish that there was a way of peeking at the stack contents with out having to go . . . SK-FORTH79 has just such a word, S. , called "stack print". Try the following:

```
11  23  34 (ret) OK
S.  (ret) BOT> 11 23 34 <TOP OK
```

The stack still contains the numbers 12 23 34, S. just prints and labels a copy of the stack for your inspection. Print the stack with . . . to prove it for yourself. The non destructive stack print can prove invaluable when debugging programs.

There is still one minor inconvenience, how do you empty the stack quickly? You may have found yourself going . . . trying to get rid of all the stack numbers but an easier way is to execute the word ABORT . Try for yourself and see what happens.

PICK and ROLL

The word PICK will use the top of the data stack as an index into the stack and obtain a copy of the selected stack number and move it to the top of the stack where it will replace the index. Example:

```
51 52 53 54 4 PICK (ret) OK
S.  (ret) BOT> 51 52 53 54 51 <TOP OK
```

The word ROLL is a general purpose stack "rotate". ROLL will rotate the stack to the depth specified by the top of the stack. Verify that the number from the last example are still on the stack and then try the following example.

```
S.  (ret) BOT> 51 52 53 54 51 <TOP OK
4 ROLL (ret) OK
S.  (ret) BOT> 51 53 54 51 52 <TOP OK
```

Note that the top stack number, 4, which gave the depth to ROLL was lost in the same way as the index used with the word PICK.

SINGLE PRECISION NUMBERS

All of the preceding examples used 16 bit single precision integer numbers. Using 16 bits we can represent signed numbers from -32,768 through +32,767 or unsigned integers 0 through 65,535. If you wish to print the stack as an unsigned integer you would use the word U. "U-dot" or unsigned print. Try the following:

```
40000 S.  (ret) BOT> -25536 <TOP OK
U.  (ret) 40000 OK
-1 U.  (ret) 65535 OK
```

It is left to the programer to decide whether operations are with signed or unsigned integers.

DOUBLE PRECISION NUMBERS

FORTH can also routinely handle 32 bit double precision numbers. With 32 bits we can represent signed numbers from -2,147,483,648 through 2,147,483,647 or unsigned numbers 0 through 4,924,967,296.

Double precision numbers are entered by including a decimal point with the number. A double precision number takes up two stack locations with the most significant part on the top of the stack. There is a complete set of stack manipulation operators or words to go along with the double precision numbers. Try the following:

```
60000.  S.  (ret) BOT> -5536 0 <TOP OK
D.  (ret) 60000 OK
80000.  S.  (ret) BOT> 14464 1 <TOP OK
D.  (ret) 80000 OK
```

As you can see the stack print operator S. is not of much use in displaying double precision numbers. To look at our double precision numbers we will have to use the double precision print operator D. "D-dot".

See if you can discover the purpose of the double precision words D+ , D- , 2SWAP , 2DUP , 2OVER , and 2ROT by making up examples of your own and using D. to look at the stack. If you get stuck then look in the glossaries. Remember, you must include a decimal point when entering double precision numbers and use the word D. to print them. Later we will show you how double precision numbers can be used to perform calculations with real (decimal) numbers.

INTRODUCTION TO COLON DEFINITIONS

All programming in FORTH consists of defining new FORTH words. New FORTH words are defined by means of colon definitions. Colon definitions have the form

: <name> .... ;

where <name> is the name of the new FORTH word to be added to the dictionary,
and .... is a series of existing FORTH words which produce the desired result.

Our first FORTH program is going to be called TWOBEEPS. The program TWOBEEPS will produce two beep sounds from the SYM-1 beeper. To write the program TWOBEEPS construct a colon definition to create a new FORTH word TWOBEEPS. The body of the colon definition (between the name and the semicolon) must contain existing FORTH words which will produce the desired action. Already compiled into the FORTH vocabulary is the word BEEP. When BEEP is executed the SYM-1 beeper will sound once (try it). Here is the program:

: TWOBEEPS BEEP BEEP ; (ret) OK

You will not hear anything when you hit return, because the new word is being created and compiled into the dictionary. To test the program type:

TWOBEEPS (ret) OK

Now you should hear two beeps!! But the two beeps are too close together! What is needed is a delay between the beeps. But first get rid of your old program by forgetting it.
Type the following:

FORGET TWOBEEPS (ret) OK

If you enter a colon definition with an error you will find that the FORTH system will not let you FORGET it. This is a built in feature that prevents the accidental execution of a definition which is incomplete or faulty. If you find that you cannot forget a word that

you know is present then execute the word SMUDGE first and try again. Look up the word SMUDGE in the glossaries for more details.

Here is the delay program:

: DELAY 10000 0 DO LOOP ; (ret) OK

The DELAY program is just an empty FORTH DO loop. To set up a FORTH DO loop, first put the final count (10000 in this case) and then the starting count (0 in the example) on the parameter stack. When executed FORTH will do everything between the DO and the LOOP, increase the loop counter, and compare it to the final count. If the final count has been reached we exit the loop and continue with the next FORTH word. If the final count has not been reached the contents of the loop will be repeated (loop contents are between the DO and LOOP). In our example there is nothing between the DO and LOOP so FORTH just counts from 0 to 10000 and then exits the loop. The DELAY program will produce a pause of about one second.

The corresponding loop in BASIC would be written as follows:

```
FOR I=0 TO 9999:NEXT I
```

The only difference is that in BASIC this loop would produce a delay of about 10 seconds. High level FORTH is about 10 times faster than BASIC.

Now we can write a better TWOBEEPS:

: TWOBEEPS BEEP DELAY BEEP ; (ret) OK

Test the program. Is the delay about one second? Now for a more sophisticated program. This one will be called MULTIBEEP.

: MULTIBEEP 0 DO BEEP DELAY LOOP ; (ret) OK

This program has a DO loop which is not empty. Each time the loop is executed there will be a beep and a delay. Note that the definition does not contain the upper loop limit. It is not required, because the upper limit will be put on the stack before MULTIBEEP is executed.

Try the following:

10 MULTIBEEP (ret) OK

Ten beeps should be heard, separated by one second pauses. Bet you never thought the built-in sound generators would come in this handy!

The next example will involve some number crunching, but first a bit more information about DO loops is required. Construct the following colon definition and verify that it works as shown below.

```
: TEST 0 DO I . LOOP ; (ret) OK
7 TEST (ret) 0 1 2 3 4 5 6 OK
```

The FORTH word I gets the current value of the loop counter and pushes it onto the parameter stack. The print number command (.) prints the top of the stack. Thus when 7 TEST is executed we see the numbers 0 thru 6. Note that 7 is not printed because the loop counter is incremented after each pass through the loop and compared to the loop limit. If the loop limit has been reached the loop is not repeated, it just passes on to the next FORTH word in the definition.

Try the following:

```
FORGET TEST (ret) OK
: TEST CR 0 DO I . CR LOOP ; (ret) OK
10 TEST (ret)          (output not shown)
```

The colon definition above uses the new FORTH word CR. CR just outputs a carriage return. When 10 TEST is executed you should see the numbers 0 through 9 printed down the screen.

By now you probably have quite a bit of clutter added to the dictionary. To see what has been added type the following:

```
VLIST (ret)
```

You can stop the VLIST by hitting the break key on your terminal. To continue with the VLIST type SPACE. To exit the VLIST type RETURN after the break key has been hit. You can get rid of some of the mess you have created by typing:

```
FORGET <name> (ret) OK
```

FORTH will then forget all words above and including <name>. You can also type

```
COLD (ret) OK
```

and the system forgets all of the words you have created and does a cold start.

The promised number crunching program will be to compute a table of squares. Here's the program:

```
: TABLE CR 10 0 DO I . I DUP * . CR LOOP ; (ret) OK
```

Test the program. You should see the numbers 0 through 9 and their squares listed down the screen. Try the following problems:

1) Construct a FORTH colon definition that produces a table of squares of any desired length. For example, to get a table of the squares 0 through 99 you should only have to type 100 TABLE (ret).

2) Modify the above program to produce both squares and cubes.

Perhaps you would like to format the table produced above. That is to say, have the numbers come out in fields of fixed width. Try the following definition:

```
: TABLE CR 0 DO I 8 .R I DUP * 8 .R CR LOOP ; (ret) OK
```

Test the definition by typing 10 TABLE (ret). The table should be as before except that the table entries should be right justified in fields of width 8.

The FORTH expression 8 .R prints the top of the stack in a field of width 8. If a field of width 10 was desired, then use 10 .R to print the top of the stack.

The only thing we are still missing from our programs is output messages that will identify the numbers that we are printing. Try the following definition:

```
: HEAD CR 2 SPACES ." NUMBER" 2 SPACES ." SQUARE" ; (ret) OK
HEAD (ret)
  NUMBER  SQUARE OK
```

Execution should produce the results shown above. The colon definition for HEAD uses two new FORTH words. SPACES will print the number of spaces that are specified by the top of the stack; the top of the stack is lost. The FORTH word (.") is the print message command. It will print all the characters that follow up to the next (") mark. Note that (.") must be followed by a blank which is not considered part of the message. Now let's combine HEAD with TABLE to produce SQUARE :

```
: SQUARE HEAD TABLE ; (ret) OK
4 SQUARE (ret)
  NUMBER  SQUARE
       0       0
       1       1
       2       4
       3       9 OK
```

When SQUARE is executed you should get the results shown above. Here are a couple of simple problems for you to try:

1) Modify either HEAD, TABLE, or both so that the numbers come out centered under the column headings.

2) Write a program (colon definition) which will produce a table of squares and cubes with headings.

So far, none of the programs that have been written has required user input while it is running. The next program will prompt the user to enter two numbers, X and Y, compute their product, and print the answer. To get input from the user the hi-level FORTH system words QUERY and INTERPRET will be used. The word QUERY will accept up to 80 characters of input from the terminal. The word INTERPRET will take the characters in the input buffer and execute them one at

a time. If the input buffer contains a number it will push that number onto the parameter stack. Here then, is our INPUT command:

```
: INPUT QUERY INTERPRET ; (ret) OK
```

Test the INPUT routine as shown below:

```
INPUT (ret) 456 (ret) OK
```

After the first return above, the cursor just sits and waits for input from the terminal. The number 456 is typed followed by return and FORTH replies with OK. We should find that 456 is on the parameter stack.

```
. (ret) 456 OK
```

That's where it was all right! In fact, this input routine is probably too powerful for the job at hand. INPUT will accept anything at all; numbers, words, even a colon definition. Try the following and see what we mean:

```
INPUT (ret) 123 BEEP 456 BEEP CR ." FANTASTIC" (ret)
FANTASTIC OK
```

You should hear two beeps and then see the FANTASTIC as shown above. This input command will even accept a colon definition and compile it into the dictionary while "your" program is running!!! This means that a FORTH program can accept as input another FORTH program. The implications are simply mind bending!! By the way, the numbers 123 and 456 should be on the stack. Type the following to verify that they are there:

```
. . (ret) 456 123 OK
```

Recall that the program must get a value of X and a value of Y, form the product, and print the answer. The program will be written in small pieces and the pieces then put into a single colon definition that will produce the desired result. First type COLD (ret) to get a fresh start.

```
: INPUT QUERY INTERPRET ; (ret) OK
: GETX CR ." ENTER VALUE OF X " INPUT ; (ret) OK
: GETY CR ." ENTER VALUE OF Y " INPUT ; (ret) OK
: ANSWER CR ." THE PRODUCT IS " * . ; (ret) OK
: MULTXY GETX GETY ANSWER ; (ret) OK
```

Test the program and verify the results shown below.

```
MULTXY (ret)
ENTER VALUE OF X 222 (ret)
ENTER VALUE OF Y 111 (ret)
THE PRODUCT IS  24642 OK
```

With the examples above as a guide you should now be able to

construct simple programs of your own. There is a lot more to FORTH than has been presented in this section. You have only seen the "tip of the iceberg". The next few sections of this manual will consist of descriptions and examples illustrating more exciting features of FORTH.

NUMBER BASES

SK-FORTH79 comes equipped to function in two number bases. These bases are DECIMAL and HEX (shortly we will see how to create others). After a cold or warm start the system number base will be DECIMAL. FORTH's standard base is decimal and if you want to form good programming habits all your routines should restore BASE to DECIMAL. To put the system into HEX mode simply type:

```
HEX (ret) OK
```

Now try the following:

```
1A E9 + . (ret) 103 OK
103 E9 - . (ret) 1A OK
2B 1E * . (ret) 50A OK
1ABC AA / . (ret) 28 OK
```

To convert from HEX to DECIMAL or vice versa simply switch bases in "midstream".

```
HEX 400 DECIMAL . (ret) 1024 OK
DECIMAL 256 HEX . (ret) 100 OK
```

MEMORY OPERATIONS

By putting FORTH into the HEX mode and using the memory operations described below you should never have to use your SUPER-MON monitor routines again.

First the store word operation.

```
!       Store 16 bit word located top-1 at address on top.

    n    addr    !   ->    ----
```

To test this memory operation store some 16 bit words in the unused memory at $6600 and then use the monitor to verify that the operation has been carried out.

```
HEX 1122 6600 ! (ret) OK
```

This should store a 22 at $6600 and an 11 at $6601. Use Control S or MON (ret) to verify this. To get back to FORTH use NEXT 0 (ret).

The fetch 16 bit word operation.

@ Replace word address on top by word contents.

addr @ -> n

To verify the word fetch operation recover the number previously stored at $6600 by using @ .

HEX 6600 @ . (ret) 1122 OK

FORTH also has operators for storing and fetching 8 bit bytes.

Store byte operation

C! store low 8 bits of top-1 at address on top

b addr C! -> ----

Fetch byte operation.

C@ replace address on top by its (8 bit) contents

Verify the above by trying the following:

HEX (ret) OK
AA 6600 C! (ret) OK
BB 6601 C! (ret) OK
6600 C@ . (ret) AA OK
6601 C@ . (ret) BB OK

There is a special FORTH operator that combines word fetch and print number. It is called print contents.

? Print 16 bit word contents of address on top

addr ? -> ----

Test ? using the following:

HEX 2BCD 6600 ! (ret) OK
6600 ? (ret) 2BCD OK

The colon definition for ? would simply be : ? @ . ; Try the following FORTH verify command:

HEX (ret) OK
: VERIFY DUP CR U. DUP 8 + SWAP DO I C@ 3 .R LOOP ;

Test the VERIFY command as shown below.

8000 VERIFY (ret)
8000 4C 7C 8B 20 FF 80 20 4AOK

Problem: Write a colon definition called DUMP which uses VERIFY in a loop to produce a hex dump of desired memory locations.

CONSTANTS AND VARIABLES

Before beginning this section type COLD (ret) to set the system in a known state.

FORTH has two defining words, VARIABLE and CONSTANT, which are used to create named data types. Up to this point all operations were carried out with unnamed data which were stored on the stack. It was necessary to keep careful tabs on the stack order to get the results that were desired.

The FORTH word CONSTANT is used to create a named data quantity whose value is expected to remain fixed for its lifespan. To create the constant PIE whose value is 31416 type the following:

31416 CONSTANT PIE (ret) OK

This causes FORTH to create an entry in the dictionary called PIE (check with VLIST if you like) and to store the value 31416 just after this entry. When a constant is executed its value is pushed onto the parameter stack. Get the value of PIE back as shown below.

PIE . (ret) 31416 OK

Once created a constant will retain its value and is not easily changed by the novice FORTH programmer. Try the following:

PIE 1416 - . (ret) 30000 OK
PIE 2 / . (ret) 15708 OK

A constant can be used just like a regular 16 bit number. You need only use its name and its value is automatically pushed to the parameter stack.

The word VARIABLE is used to create a named data type whose value is expected to change from time to time during its lifespan. When a variable is created its initial value is not specified. To create a variable called RADIUS whose initial value is 16 type the following:

VARIABLE RADIUS 16 RADIUS ! (ret) OK

The initial value of a variable must be assigned after it is created. When a variable is executed its address (and not its value) is pushed onto the parameter stack. Thus to recover the actual value of a variable you must use its name and the memory fetch operator (@). Try the following:

RADIUS @ .(ret) 16 OK
RADIUS ? (ret) 16 OK

To find the area of a circle of radius 16 the constant PIE of the previous example can be used as follows:

```
RADIUS @ DUP * PIE 10000 */ . (ret) 804 OK
```

It is left as an exercise to the reader to look up in the glossary and find out what */ does.

Programming exercise:

Write a program that will prompt for the radius of a circle and store it in the variable RADIUS and then compute the area and circumference and print them out with identifying labels. Hint: To change the value of the variable RADIUS to 12 type 12 RADIUS ! .

NUMBER BASES REVISITED

The current number base that the FORTH system is using is stored in the system variable called BASE. The FORTH system comes with two words which can be used to change the variable BASE to either 10 or 16. These words are, of course, DECIMAL and HEX, which have already been used. If you wish, you can add your own words which will change the value of BASE. Believe it or not, this is all that is required to switch the entire FORTH system so that all calculations and results are in the new base. FORTH can work in any base from 2 through 36. Here are the two other standard bases that most FORTH programmers add to their system.

```
: OCTAL 8 BASE ! ; (ret) OK
: BINARY 2 BASE ! ; (ret) OK
```

SYSTEM RECONFIGURATION

The delivered SK-FORTH79 systems comes with high memory set at $6000 two 1K screen buffers and 8 simulated disk ram screens from LO=$6000 throught HI=$8000. The Getting started section provided some words to convert to 3 typical systems. Below is a general purpose RECONFIG.

```
    FORTH DEFINITIONS HEX
  : RECONFIG   HI ! DUP LO !       ( SET DISK IN RAM SCREENS )
               80 - DUP ' LIMIT ! ( RESET CONSTANT LIMIT       )
               DUP 2E +ORIGIN !    ( RESET IN BOOT UP AREA ALSO )
               OVER B/BUF 4 + *    ( COMPUTE TOTAL BUFFER SIZE  )
               - ' FIRST !         ( RESET CONSTANT FIRST       )
               ' NBUF ! COLD ;     ( RESET NBUF AND COLD START  )
```

The word RECONFIG requires three parameters as shown below.

```
<number of buffers> <value of LO> <value of HI> RECONFIG -> ----
```

To reconfigure for a 24K cassette system with 4 disk ram screens and two disk buffers type:

```
HEX 2 5000 6000 RECONFIG (ret)
BOOT (ret)
```

Don't forget to save the reconfigured system.

SUMMARY OF STACK OPERATORS

SWAP Exchange top two stack items.
```
n1  n2  SWAP  ->  n2  n1
```

DUP Duplicate top of stack.
```
n  DUP  ->  n  n
```

?DUP Duplicate top of stack only if nonzero.
```
n  ?DUP  ->  n  (n)
```

DROP Discard the top of the stack.
```
n  DROP  ->  ----
```

OVER Copy the second stack entry to the top of the stack.
```
n1  n2  OVER  ->  n1  n2  n1
```

ROT Rotate third entry to the top of the stack.
```
n1  n2  n3  ROT  ->  n2  n3  n1
```

PICK Pick a copy of the n-th stack value to the top.
```
n  PICK  ->  n-th
```

ROLL Rotate the n-th item to the top of the stack.
```
. . . n  ROLL  ->  . . .  n-th
```

2SWAP Exchange top two double numbers.
```
d1  d2  2SWAP  ->  d2  d1
```

2DUP Duplicate top double number.
```
d  2DUP  ->  d  d
```

2DROP Discard the top double number.
```
d  2DROP  ->  ----
```

2ROT Rotate third double number to the top.
```
d1  d2  d3  2OVER  ->  d2  d3  d1
```

2OVER Copy the second double number to the top.

>R Move top data stack number to return stack.
```
n  >R  ->  ----
```

R> Retrieve top of return stack to the top of data stack.
```
----  R>  ->  n
```

R@ Copy top of return stack to the top of the data stack.
```
----  R@  ->  n
```

DEPTH Count 16 bit entries in data stack and leave on top.
```
----  DEPTH  ->  n
```

RDEPTH Count 16 bit entries in return stack and leave on top.
```
----  RDEPTH  ->  n
```

SUMMARY OF OUTPUT OPERATORS

. Print top of stack with one trailing blank.
```
n . -> ----
```

? Print contents of address on top at console.
```
addr ? -> ----
```

D. Print top double number with one trailing blank.
```
d D. -> ----
```

U. Print top of stack as unsigned 16 bit number.
```
un U. -> ----
```

.R Print second from top to right of field given by top.
```
n1 n2 .R -> ----
```

D.R Print print double number to right of field given by top.
```
d n D.R -> ----
```

S. Non destructive stack print.
```
---- S. -> ----
```

CR Output a carriage return linefeed.
```
---- CR -> ----
```

EMIT Output low 7 bits of top as an ASCII character.
```
char EMIT -> ----
```

SPACE Output one space.
```
---- SPACE -> ---
```

SPACES Output n spaces.
```
n SPACES -> ----
```

." <text>" Output inline text message to the console.
```
---- ." <text>" -> ----
```

SUMMARY OF ARITHMETIC OPERATORS

+ Add top two single numbers on the stack.
```
n1 n2 + -> sum
```

D+ Add top two double numbers and leave as dsum.
```
d1 d2 D+ -> dsum
```

- Subtract top two single numbers and leave as n.
```
n1 n2 - -> n
```

D- Subtract top two double numbers and leave as d.
```
d1 d2 D- -> d
```


1+ Add one to the top number on the stack.
```
n 1+ -> n+1
```

1- Subtract one from the top number on the stack.
```
n 1- -> n-1
```

2+ Add two to the top number on the stack.
```
n 2+ -> n+2
```

2- Subtract two from the top number on the stack.
```
n 2- -> n-2
```

***** Multiply the top two numbers and leave as n3
```
n1 n2 * -> n3
```

/ Divide second from top by top and leave n3=n1/n2
```
n1 n2 / -> n3
```

U* Multiply unsigned single numbers leaving unsigned double result.
```
un1 un2 U* -> ud
```

MOD Leave remainder on division n1/n2 with sign of n1.
```
n1 n2 MOD -> rem
```

/MOD As above but leave quotient too.
```
n1 n2 /MOD -> rem quot
```

***/MOD** Multiply then divide with double precision intermediate.
```
n1 n2 n3 */MOD -> rem quot
```

***/** Like */MOD above but leave quotient only.
```
n1 n2 n3 */ -> quot
```

U/MOD Divide double number by single number leaving remainder and quotient.
```
d n U/MOD -> urem uquot
```

M* Multiply signed single numbers leaving signed double result.
```
n1 n2 M* -> d
```

M/ Divide double number by single number leaving signed remainder and quotient.
```
d n M/ -> rem quot
```

M/MOD Divide unsigned double number by unsigned single number leaving unsigned single remainder and unsigned double quotient.
```
ud un M/MOD -> urem udquot
```

MAX Leave greater of two single numbers as n.
```
n1 n2 MAX -> max
```

DMAX Leave greater of two double numbers as dmax.
```
d1 d2 DMAX -> dmax
```

MIN Leave the smaller of two single numbers as min.
```
n1  n2  MIN  ->  min
```

DMIN Leave the smaller of two double numbers as dmin.
```
d1  d2  DMIN  ->  dmin
```

ABS Leave absolute value of number on top as abs.
```
n  ABS  ->  abs
```

DABS Leave absolute value of double number on top as dabs.
```
d  DABS  ->  dabs
```

NEGATE Leave twos complement of single number on top.
```
n  NEGATE  ->  -n
```

DNEGATE Leave twos complement of the double number on top.
```
d  DNEGATE  ->  -d
```

AND Bitwise logical and of top two 16 bit numbers.
```
n1  n2  AND  ->  and
```

OR Bitwise logical or of top two 16 bit numbers.
```
n1  n2  OR  ->  or
```

XOR Bitwise logical exclusive or of top two 16 bit numbers.
```
n1  n2  XOR  ->  xor
```

SUMMARY OF MEMORY OPERATORS

C@ Replace address on top with least significant byte.
```
addr  C@  ->  byte
```

@ Replace address on top with 16 bit value at address.
```
addr  @  ->  n
```

2@ Replace address on top with 32 bit value at address.
```
addr  2@  ->  d
```

C! Store least significant byte of n at address on top.
```
n  addr  C!  ->  ----
```

! Store 16 bit value n at address on top.
```
n  addr  !  ->  ----
```

2! Store 32 bit value d at address on top.
```
d  addr  2!  ->  ----
```

+! Add n to number stored at address on top.
```
n  addr  +!  ->  ----
```

CMOVE Move n bytes starting at addr1 to memory at addr2 if n>0.
```
addr1  addr2  n  CMOVE  ->  ----
```


MOVE Move n 16 bit numbers starting at addr1 to addr2 if n>0.
```
addr1  addr2  n  MOVE  ->  ----
```

FILL Fill n bytes in memory with byte beginning at addr if n>0.
```
addr  n  byte  FILL  ->  ----
```

ERASE Fill n bytes of memory starting at addr with nulls.
```
addr  n  ERASE  ->  ----
```

BLANKS Fill n bytes of memory starting at addr with blanks ($20).
```
addr  n  BLANKS  ->  ----
```

SUMMARY OF INPUT OPERATORS

KEY Get console key pushed without echo and leave on stack.
```
----  KEY  ->  char
```

EXPECT Read n characters from terminal to address.
```
addr  n  EXPECT  ->  ----
```

QUERY Read up to 80 characters from terminal to input buffer.
```
----  QUERY  ----
```

WORD Read next word from input stream using char as delimiter.
```
char  WORD  ->  addr
```

COMPARISON OPERATORS

The comparison operators below are used in conjunction with the control structures IF , UNTIL , and WHILE which are discused in the next section. Each comparison operator leaves either a false flag (16 bit 0) or a true flag (16 bit 1). For the purposes of the control structures any non zero quantity is considered true.

```
n1   n2   <    ->  flag   True if n1 is less than n2.

n1   n2   =    ->  flag   True if n1 is equal to n2.

n1   n2   >    ->  flag   True if n1 is greater than n2.

     n    0<   ->  flag   True if n is less than 0.

     n    0=   ->  flag   True if n is equal to 0.

     n    0>   ->  flag   True if n is greater than 0.

un1  un2  U<   ->  flag   True if unsigned number un1 is less
                          than unsigned number un2.

     n    NOT  ->  flag   True if n is equal to 0.
```

The remaining comparison operators are for double numbers.

d1	d2	D<	->	flag	True if d1 is less than d2.
d1	d2	D=	->	flag	True if d1 is equal to d2.
d1	d2	D>	->	flag	True if d1 is greater than d2.
	d	D0<	->	flag	True if d is less than double 0.
	d	D0=	->	flag	True if d is equal to double 0.
	d	D0>	->	flag	True if d is greater than double 0.
ud1	ud2	DU<	->	flag	True if unsigned double number d1 is less than unsigned double number d2.

CONDITIONALS AND INDEFINITE LOOP OPERATORS

All FORTH conditionals make a decision based on a flag left on the top of the data stack. In most cases this flag has been computed by one of the comparison operators of the previous section. The conditional words IF , ELSE , and THEN are used as follows:

```
. . . flag   IF . . . true part. . . THEN . . . words
```

Try the following example:

```
: KEYCHECK KEY DUP 48 > SWAP 57 < AND IF BEEP THEN ; OK
```

When KEYCHECK is executed it waits for a console key to be pushed. If the key pushed is a digit 0 through 9 you will hear a beep, otherwise there will be no sound. The entire expression " DUP 48 < SWAP 57 > AND " computes the "flag" that IF tests.

The second form of this construct is shown below.

```
. . flag  IF . . true part . . ELSE . . false part . . THEN . .
```

Here is an example to try:

```
: CHK1 KEY 32 < IF ." UNPRINTABLE " ELSE ." PRINTABLE " THEN ; OK
```

The word CHK1 waits for a key to be pushed. If the key pushed is a control key the message UNPRINTABLE is typed. If any other key is pushed the message PRINTABLE is typed. Here is a short puzzle for you to solve- Why does the definition CHK2 given below do exactly the same thing?

```
: CHK2 KEY 32 < IF ." UN" THEN ." PRINTABLE " ; OK
```

It should be noted that the words IF , ELSE , and THEN can only be used within a colon definition.

Indefinite and infinite loops can be constructed from the words BEGIN , AGAIN , WHILE , REPEAT , and UNTIL as shown in the following examples.

To construct an infinite loop, ie one that will never finish, use the structure:

```
. . . BEGIN . . . words . . . AGAIN . . .
```

At first glance it would seem that this construct is not of much use as you could never stop the program. Try the following and you will probably change you mind.

```
: SEE-STACK BEGIN CR QUERY INTERPRET CR S.  AGAIN ; OK
```

Now execute SEE-STACK. You are traped, or so it appears, in an infinite loop that gets input from the terminal, interprets it and then does a non destructive stack print. Try keying in some arithmetic operations and see if you like this definition. There are two ways out of this particular infinite loop, the first is to make an error, the second is to type QUIT. This only works because the loop includes the word INTERPRET.

If you are not sold on infinite loops try the following construct for indefinite loops. An indefinite loop is one that continues until a certain condition is satisfied.

```
. . . BEGIN . . . words . . . flag UNTIL . .
```

This loop will continue to branch back to just after BEGIN until the flag is true. Try the following:

```
: COUNT-DOWN 10 BEGIN BEEP CR DUP .  1- DUP 0= UNTIL
             CR ." BLAST OFF " DROP ;
```

The word DROP at the end of the definition removes the 0 which would otherwise be left on the stack. Here is another form of the indefinite loop:

```
. . . BEGIN . . words . . flag WHILE . . true part . . REPEAT . .
```

In this looping structure, as long as the flag is true the part between WHILE and REPEAT is executed. When REPEAT is reached there is an unconditional branch to just after BEGIN. If the flag is false, execution continues just ahead of REPEAT. Below is an example which uses this structure.

```
: COUNT-UP 0 BEGIN CR DUP .  DUP 10 = NOT WHILE 1+ BEEP REPEAT
                ." DONE " DROP ;
```

You will hear the part between WHILE and REPEAT execute because of the BEEP. If you haven't already noticed, you may split a long defintion and put it on two lines and every thing works just fine, as long as there are no mistakes. The following examples combines some of the ideas of the last two sections. It can be entered line after

line only if you make zero typing mistakes. You may prefer to leave this example until you have studied the section on the EDITOR.

```
: KEYTEST   BEGIN    KEY
            DUP 32  < IF  ." CONTROL CHARACTER "    ELSE
            DUP 48  < IF  ." PUNCTUATION MARK  "    ELSE
            DUP 65  < IF  ." IT'S A NUMBER     "    ELSE
            DUP 91  < IF  ." CAPITAL LETTER    "    ELSE
            DUP 96  < IF  ." PUNCTUATION MARK  "    ELSE
            DUP 122 < IF  ." LOWER CASE LETTER "    ELSE
                          ." PUNCTUATION MARK  "
            THEN  THEN  THEN  THEN  THEN  THEN
                  CR  13 = UNTIL ;
```

When you execute KEYTEST any key pushed on the console will be identified according to its class. This loop will continue until the RETURN key is pushed.

It should again be noted that the words BEGIN , WHILE , REPEAT , AGAIN , and UNTIL can only be used within colon definitions.

FINITE LOOPS

FORTH also has a construct similar to BASIC's FOR . . . NEXT loop and FORTRAN's DO loop. The first form of the FORTH's DO . . . LOOP is shown below:

limit index DO . . . words . . . LOOP . . .

This structure executes the words between DO and LOOP for each count from index to limit incrementing by 1. The loop counter is stored on the return stack and can be copied to the data stack by using the word I. Example:

```
: COUNT-UP  10 0 DO I .  LOOP ; OK
COUNT-UP (ret) 0 1 2 3 4 5 6 7 8 9 OK
```

Note that the loop is not executed when I is equal to limit. The loop counter is incremented by 1 and checked against limit at LOOP and control is passed to just after LOOP if the limit has been reached or exceeded. If the limit has not been reached then control transfers back to just after DO. DO . . . LOOP's can also be nested and the words J , and K are available to access the second and third outer loop counters respectively. Try the following:

```
: LOOPING 3 0 DO CR CR 4 0 DO CR 5 0 DO I J K + + .  LOOP LOOP LOOP ;
```

When you execute LOOPING you should see three rectangular arrays of numbers. It is left as an exercise to verify the operation of LOOPING.

To construct count-down loops and loops which increment by more than 1 use the following sturcture.

limit index DO . . . words . . . n +LOOP . . .

In this form the value of n is added to the loop counter at +LOOP and the result compared to limit. If n is greater than zero, the branch back to DO occurs until the the new index is equal to limit or greater than limit. If n is less than zero, the branch back to DO occurs until the new index is less than the limit. Study the following examples:

```
: UPBY2 10 0 DO I .  2 +LOOP ; OK
UPBY2 (ret) 0 2 4 6 8 OK

: DOWNBY2 0 10 DO I .  -2 +LOOP ; OK
DOWNBY2 (ret) 10 8 6 4 2 0 OK
```

In the example DOWNBY2 where n is negative the loop is executed until the loop counter is less than the limit. Whereas in the example UPBY2 where n is positve the loop is executed until the loop counter is equal to or greater than the limit.

The FORTH word LEAVE can be used to prematurely exit from a DO . . . LOOP structure. Below is an example of a very long loop which can be terminated early if the user pushes the BREAK key on the console.

```
: LONG 10000 0 DO CR I .  ?TERMINAL IF LEAVE THEN LOOP ; OK
```

The word LEAVE sets the loop counter equal to the limit so that when +LOOP or LOOP is next encountered the DO . . . LOOP is terminated. ?TERMINAL leaves a true flag on the stack if the BREAK key has been pushed, otherwise it leaves a false flag. After pushing the BREAK key, push RETURN to abort and SPACE if you wish to continue program execution.

SUMMARY OF CONDITIONAL AND LOOP OPERATORS

DO... Set up loop given index limit and start on the stack.
limit index DO...

...LOOP Increment loop index by 1 and repeat from DO until limit is reached.
...LOOP

...+LOOP Increment loop index by stack value and repeat from DO if index >= limit (n>0), or < limit if (n<0).
... n +LOOP

I Place current loop index on the data stack.
---- I -> n

J Place second outer loop index on the data stack.
---- J -> n

K Place third outer loop index on the data stack.
---- K -> n

LEAVE Terminate loop at next LOOP or +LOOP by setting index equal to limit.
---- LEAVE -> ----

IF...THEN If flag is true execute part between IF and THEN.
flag IF...true...THEN . . .

IF... If flag is true execute part between IF and ELSE,
...ELSE... If flag is false execute part between ELSE and THEN,
...THEN in both cases resume execution after THEN.
flag IF...true...ELSE...false...THEN...

BEGIN... Mark the start of a repetative execution sequence.
BEGIN...words...AGAIN , or UNTIL , or WHILE

...AGAIN Mark the end of an infinite loop.
BEGIN...words...AGAIN...

...UNTIL Loop back to just after begin if flag is false.
BEGIN...words...flag UNTIL...

...WHILE... If flag is true execute words between WHILE and
...REPEAT REPEAT and pass control back to just after BEGIN.
...BEGIN...words...flag WHILE...words...REPEAT...

EXIT Terminate execution of colon definition.
Not valid within a DO...LOOP
---- EXIT -> ----

FORGETING THE UNFORGETABLE

If you find that you cannot FORGET a word definition it may be because the definition was incomplete or faulty in some way. When this happens a bit in the name field called the smudge bit is set to prevent its execution. This also prevents FORGET from finding it even though you may see it in the VLIST. If the word you are trying to FORGET was the last one defined type the word SMUDGE and then try again.

It is also possible to FORGET words that are in the protected dictionary. To do this you must reset the system variable FENCE. Suppose the word JUNK was locked in the protected dictionary. The following sequence would remove junk and everything defined after JUNK.

```
' JUNK NFA FENCE ! FORGET JUNK (ret) OK
```


THE EDITORS

Up to this point, each of the FORTH definitions entered at the console "disappear" when RETURN is pushed. They may stay on the screen for a while but there is no way to recover them for correction or even to look at once they are off the top of the CRT. This is because each definition is compiled into the dictionary as it is entered. This is no problem as there is provision for the off line preparation of FORTH definitions.

SK-FORTH79 has two editors available for preparing FORTH source definitions:

1. A fig (FORTH Interest Group) style editor.
2. A visual screen editor. The source for the KTM-2 version of this editor is in the appendix and must be entered by the user.

Both the fig and visual editors work with a unit of information called the "screen". A screen is 16 lines of 64 characters or 1024 bytes of information. To see what a screen looks like type:

```
1 LIST (ret)
```

In SK-FORTH these screens are automatically fetched and returned to the mass storage device. For those without a disk system this mass storage is simulated in ram and may consist of from four to eight screens or blocks (or more if you have the extra memory). The screens of information (somtimes called disk blocks) are fetched and and updated to mass storage through two 1K (1024 byte) disk buffers. There could be as many as two screens of information (one for each buffer) resident in memory at any one time. The process of fetching and updating is done automatically and is often referred to as virtual memory.

Two important words to remember with regard to the screen buffers are:

SAVE-BUFFERS and EMPTY-BUFFERS

The word SAVE-BUFFERS forces the updating of all modified screen buffers to the mass storage device. The word EMPTY-BUFFERS will clear the two memory resident screen buffers to blanks. This should be used with caution. One application would be when disk users wish to temorarily switch to cassette operation. The sequence would be as follows:

```
SAVE-BUFFERS  ( Update all current screens to the old file )
0 DISK !      ( Switch to cassette mass storage )
EMPTY-BUFFERS ( Clear all buffers to blanks )
```

If we did not SAVE-BUFFERS first our disk mass storage would not be updated. If we did not EMPTY-BUFFERS some of the old disk buffers could get updated to the ram mass storage over writing valuble

information.

FIG STYLE EDITOR

The standard commands for the fig style editor are presented below.

CLEAR Clear indicated screen to blanks.
```
scr   CLEAR  ->   ----
```

COPY Copy entire contents of screen scr1 to screen scr2.
```
scr1   scr2   COPY  ->   ----
```

D Delete indicated line from screen.
```
line   D  ->  ----
```

EDITOR Allow access to the EDITOR commands.

E Fills indicated line of current screen with blanks.
```
line   E  ->   ----
```

FIX List indicated line and enter line edit mode.
```
line   FIX  ->   ----
```

H Hold copy indicated line at PAD.
```
line   H  ->   ----
```

I Spread and insert line at PAD at indicated line.
```
line   I  ->   ----
```

L List the screen currently stored in variable SCR .
```
----   L  ->   ----
```

LIST List indicated screen and reset value of SCR .
```
scr   LIST  ->   ----
```

P Put following text at indicated line.
```
line   P  <text>  ->   ----
```

R Return line stored at PAD to indicated line.
```
line   R  ->   ----
```

S Spread at indicated line and insert a blank line.
```
line   S  ->   ----
```

SCR Variable containing current editing screen.

T Type indicated line of current screen to console.
```
line   T  ->   ----
```

The following takes you through the preparation of a screen of FORTH source text:

Make EDITOR the CONTEXT vocabulary by typing:


```
EDITOR  (ret) OK
```

Finally clear the two screens by typing:

```
3  CLEAR  4  CLEAR  (ret) OK
```

To list screen 3 type 3 LIST (ret) . You should see a blank screen.

To relist the current screen (the one contained in SCR) you need only type L (ret) without any parameters. To enter a line into the screen use the put command, P . Try the following:

```
0 P ( TABLE OF SQUARES )
1 P   FORTH  DEFINITIONS
2 P   : HEAD CR 2 SPACES ." NUMBER"
3 P            2 SPACES ." SQUARE" ;
4 P   : TABLE CR 0 DO I 8 .R
5 P           I DUP * 8 .R CR LOOP ;
6 P   : SQUARE HEAD TABLE ;
```

(Note: The (ret)'s and OK's are not shown above.) Now type L (ret) and you should see:

```
Screen 3  3 hex
 0 ( TABLE OF SQUARES )
 1   FORTH  DEFINITIONS
 2   : HEAD CR 2 SPACES ." NUMBER"
 3            2 SPACES ." SQUARE" ;
 4   : TABLE CR 0 DO I 8 .R
 5           I DUP * 8 .R CR LOOP ;
 6   : SQUARE HEAD TABLE ;
 . . .
```

Once you have the SQUARE program entered in screen 3 correctly you can compile it into the dictionary. To compile screen 3 type the following:

```
3  LOAD  (ret) OK
```

Test the function and use VLIST to verify its existence. If you get a LOAD error type WHERE (ret) and you will be notifed as to the location of your error. If you have a mistake use the FIX command to correct it.

THE VISUAL EDITOR

The visual editor ressembles the fig editor except that you see all the editing functions take place before your eyes on an image of the screen. The best way to learn to use the visual editor is to experiment with the commands.

ED Make EDITOR context vocabulary and list current

screen.

line PP — Move to indicated line in screen for insertion or correction of the line.

line EE — Erase indicated line from screen.

line SS — Spread at indicated line.

line DD — Delete indicated line and close up space.

line II — Spread at indicated line and move in for insertion of a new line.

LL — List the current screen.

scr LI — List the indicated screen.

ln1 ln2 MM — Move line ln1 to line ln2.

ln1 ln2 CC — Copy line ln1 to line ln2.

<< — Move back one screen.

>> — Move forward one screen.

MASS STORAGE OPERATORS

All communication with the FORTH systems mass storage takes place though the disk buffers. In addition SK-FORTH as a number of words that facilitate the transfer of the simulated disk in ram screens to and from the remote controlled audio cassette recorders. All words relating to cassette and mass storage operation are grouped together here with brief descriptions for your convenience.

PUT Save screen n2 to the cassette with file ID of n1
```
n1  n2  PUT  ->  ----
```

GET Load the file with identification n1.
```
n1  GET  ->  ----
```

SGET Force load the very next tape file to screen n1.
```
n1  SGET  ->  ----
```

W/T System variable contain number of the output cassette. 0 is the standard cassette and 1 is the optional cassette.

R/T System variable containing number of the input cassette.

TAPE Set the state of cassette motor n2 to n1 where 0 is off and 1 is on.
```
n1  n2  TAPE  ->  ----
```


DISK System variable set to 0 for cassette system 1 if the HDE disk system is present.

UPDATE Mark last block referenced as modified.
```
----  UPDATE  ->  ----
```

SAVE-BUFFERS Write all blocks marked as UPDATEd to mass storage.
```
----  SAVE-BUFFERS  ->  ----
```

EMPTY-BUFFERS Mark all block/screen buffers as empty.
```
----  EMPTY-BUFFERS  ->  ----
```

BLOCK Leave memory address of requested block reading from mass storage if necessary.
```
n  BLOCK  ->  address
```

LOAD Interpret screen n and then resume interpretation of the current input stream.
```
n  LOAD  ->  ----
```

SYSSAVE Save the current FORTH system to cassette with file identification n .
```
n  SYSSAVE  ->  ----
```

SYSCOLD Locks all recently made definitions into the current FORTH system.
```
----  SYSCOLD  ->  ----
```

BOOT Initialize disk buffers after a cold start.
```
----  BOOT  ->  ----
```

R/W The standard FORTH disk read/write linkage.
```
addr  blk  flag  R/W  ->  ----
```

DRW Disk mass storage read/write linkage.

TRW Ram mass storage read/write linkage.

INDEX List first line of specified range of screens.
```
scr1  scr2  INDEX  ->  ----
```

MON Exit to the SYM/KIM monitor.
```
----  MON  ->  ----
```

A more detailed description of each of the above words is available in the glossaries at the end of the manual.

VOCABULARY OPERATORS

SK-FORTH79 has three standard system VOCABULARYs or logical groupings of words.

FORTH Contains all the basic definitions including the disk and cassette interface words.

EDITOR Contains all the commands that pertain to the fig style editor and the visual editor.

ASSEMBLER Contains all the 6502 assembler mnemonics and the assembler macros.

It is also possible to create your application vocabularies as will be demonstrated later in this section. Here is a list of the FORTH words that pertain to vocabularies.

CONTEXT System variable pointing to the vocabulary first searched for word names by the interpreter.

CURRENT System variable pointing to the vocabulary where new word definitions are to be added.

DEFINITIONS Sets the system variable CURRENT so it points to same vocabulary as the variable CONTEXT.

FORTH Execution makes CONTEXT point to FORTH.

EDITOR Execution makes CONTEXT point to EDITOR.

ASSEMBLER Execution makes CONTEXT point to ASSEMBLER.

VOCABULARY Create a new vocabulary, name follows.
---- VOCABULARY <name> -> ----

' <name> "tick", Find PFA of <name> in the dictionary and leave on the data stack.

FIND Leave the compilation address of the next word in the input stream.
---- FIND <name> -> addr

FORGET Forget all definitions back to and including <name>.
---- FORGET <name> -> ----

The two screens below create an new vocabulary called SOUND. The words in the SOUND vocabulary will use one to the shift registers in the SYM 6522 I/O chip. You will have build a one transistor amplifier to hear the sound. For more information on sound generation see Saturn Softnews Vol. 1 No. 1 .


```
Screen 4    4 hex
 0 ( VOCABULARY DEFINITION EXAMPLE-1 J.W.B.  19:01:82 )
 1 VOCABULARY SOUND IMMEDIATE ( Vocabularies are IMMEDIATE )
 2 SOUND DEFINITIONS HEX ( CONTEXT and CURRENT point to SOUND )
 3
 4 A80B CONSTANT CONTROL.REGISTER    A808 CONSTANT FREQUENCY
 5 A80A CONSTANT SHIFT.REGISTER
 6 : SHIFT SHIFT.REGISTER C! ;     : FREQ FREQUENCY C! ;
 7 : ON 10 CONTROL.REGISTER C! 0 FREQ ;
 8 : OFF 00 CONTROL.REGISTER C! ;
 9 : TONE ON OF SHIFT FREQ ;
10
11 : PHASOR FF 0 DO I TONE LOOP OFF ;      ( PHASOR SOUND EFFECT )
12
13 : TORP ON OF SHIFT                      ( TORPEDO SOUND EFFECT )
14          FF 00 DO I FREQ I DROP LOOP OFF ;
15 -->

Screen 5    5 hex
 0 ( VOCABULARY DEFINITIONS EXAMPLE-2 J.W.B.  19:01:82 )
 1
 2 VARIABLE SEED     ( RANDOM NUMBER GENERATOR rmax RND -> r? )
 3 : (RND) SEED @ 103 * 3 + 7FFF AND DUP SEED ! ;
 4 : RND   (RND) 7FFF */ ;
 5  DECIMAL
 6 : S>> ON 51 SHIFT 50 200 DO I FREQ -1 +LOOP ;
 7 : S<< ON 51 SHIFT 200 50 DO I FREQ LOOP ;
 8 : SIREN S>> S<< OFF ;
 9 : RSIREN  10 RND 2+ 0 DO SIREN LOOP ;
10
11 : RPHASOR 5 RND 2+ 0 DO PHASOR LOOP ;
12 : RTORP   5 RND 2+ 0 DO TORP LOOP ;
13 : WAR   3 0 DO RTORP RSIREN RPHASOR LOOP ;
14 FORTH DEFINITIONS
15
```

If you feel that you have the screens keyed in correctly type 4 LOAD. When the forward arrow in line 15 of screen 4 is reached compilation will automatically continue to the next screen which is 5 in this case. You pronounce --> as "next screen". If you find that you have made errors then FORGET SOUND, correct the mistakes and type 4 LOAD again. When you get a successful compilation try the following

RSIREN (ret) RSIREN ? OK

You didn't hear a thing did you! Only the message that means that RSIREN can't be found. The reason for this is that RSIREN is not in the FORTH vocabulary. RSIREN is in the SOUND vocabulary. To hear RSIREN you must make SOUND the CONTEXT vocabulary. To make SOUND the CONTEXT vocabulary simply type SOUND. Try again with:

SOUND RSIREN (ret) OK

ERROR MESSAGES

If you have the luxury of five or more simulated disk screens and would like to have error messages output instead of error numbers then type the following into screens 1 and 2. Next set WARNING to 1, i. e., 1 WARNING ! . Better save the screens to cassette so that you don't have to type them in again!! If your system has only 24K of memory you may wish to just look up the error messages here when necessary. If you have the HDE disk system just type them into screens 1 and 2 and SAVE-BUFFERS.

```
Screen  1     1 hex
 0 (  ERROR MESSAGES )
 1 (  EMPTY STACK )
 2 (  DICTIONARY FULL )
 3 (  HAS INCORRECT ADDRESS MODE )
 4 (  ISN'T UNIQUE )
 5
 6 (  DISK RANGE ? )
 7 (  FULL STACK )
 8 (  DISK ERROR )
 9
10 (  TAPE WRITE ERROR )
11 (  TAPE READ ERROR )
12
13
14
15   SK-FORTH79  2.0 COPYRIGHT SATURN SOFTWARE LIMITED


Screen   2     2 hex
 0 (  ERROR MESSAGES )
 1 (  COMPILATION ONLY, USE IN DEFINITION )
 2 (  EXECUTION ONLY )
 3 (  CONDITIONAL NOT PAIRED )
 4 (  DEFINITION NOT FINISHED )
 5 (  IN PROTECTED DICTIONARY )
 6 (  USE ONLY WHEN LOADING )
 7 (  OFF CURRENT EDITING SCREEN )
 8 (  DECLARE VOCABULARY )
 9
10 (  RELATIVE BRANCH OUT OF RANGE )
11 (  INVALID ADDRESSING MODE )
12
...
15
```


DICTIONARY OPERATORS

By now you are aware that every new word definition created is entered into the dictionary and that all the words in the context vocabulary can be listed with VLIST. If you haven't looked at the words in the EDITOR and ASSEMBLER vocabularies yet, try the following:

```
EDITOR VLIST (ret)
ASSEMBLER VLIST (ret)
```

If you get tired of waiting push the break key to halt the VLIST then push SPACE to resume or RETURN ot abort the VLIST.

Below are some of the dictionary operators that you may find useful from time to time.

VLIST List all word definitions in the context vocabulary.

DP User variable containing the address of the next memory location in the dictionary.

HERE Leave address of next available dictionary location on the data stack.

ALLOT Advance dictionary pointer by top number on data stack. The dictionary pointer is stored in DP and accessed by HERE.

, Store or compile top stack number in the next two available dictionary locations.

C, Compile the low 8 bits of top stack number into the next available dictionary location.

CREATE Create a dictionary entry for the word <name> that follows in the input stream, without allocating any parameter field. When <name> is subsequently executed, the address of <name>'s parameter field is left on the data stack.
---- CREATE <name> -> ----

' "tick" Leave the parameter field address, pfa , of the dictionary word <name>.
---- ' <name> -> pfa

CFA Convert the parameter field address, pfa , on top of the stack to the code field address, cfa.
pfa CFA -> cfa

LFA Convert the parameter field address, pfa , on top of the stack to the link field address, lfa.
pfa LFA -> lfa

NFA Convert the parameter field address, pfa , on

top of the stack to the name field address nfa.
pfa NFA -> nfa

PFA Convert the name field address , nfa , on top of the stack to the parameter field address.
nfa PFA -> pfa

ID. Print word definition name from the nfa found on top of the data stack.
nfa ID. -> ----

Again, more complete descriptions can be found in the glossaries. For more information on nfa, pfa, lfa, and cfa see the section of the manual titled "Anatomy of a dictionary entry". To illustrate the usage of some of the above words we will create a data table with 10 entries. This will be done by creating a dictionary entry and then alloting 20 bytes (10 16 bit numbers) of dictionary space following this entry. Try the following:

```
CREATE TABLE (ret) OK
20 ALLOT (ret) OK
```

We have now created a dictionary entry called TABLE and ALLOTed enough dictionary space for 10 16-bit stack numbers. Do a VLIST and see for yourself. Now enter the following to convince yourself that we have indeed created a table where we can store data.

```
: FILL-TABLE 10 0 DO I TABLE I 2* + ! LOOP ; (ret) OK
  FILL-TABLE (ret) OK
```

The data table, TABLE , will now contain the numbers 0 through 9. To print out the values stored in the table, try the following:

```
: PRINT-TABLE 10 0 DO TABLE I 2* + @ .  LOOP ; (ret) OK
  PRINT-TABLE (ret) 0 1 2 3 4 5 6 7 8 9 OK
```

If you haven't already guessed, the expression TABLE I 2* + in both FILL-TABLE and PRINT-TABLE addresses the Ith 16 bit data storage address in the dictionary space we ALLOTed after the creation of TABLE. In the next section we will generalize the concept of the data table and show to extend the FORTH compiler by creating a new defining word for VECTOR data.

EXTENDING THE FORTH COMPILER

Step right up and before your eyes we will show you how to extend the FORTH compiler by adding new defining word using the FORTH-79 Standard structure-

```
: <name> CREATE . . words . . DOES> . . words . . ;
```

First we assume that you have as part of your system the words ON ,

OFF , and TONE that were given as part of the SOUND vocabulary a while back. If you don't have them in the dictionary then go back and re-enter them. We are going to create an miniture music language. First we present the steps in reverse order (because the last step is the easiest and the first is the hardest).

Step 3 Define a word which will play the C major scale through the speaker.

```
: SCALE 1 C 1 D 1 E 1 F 1 G 1 A 1 B 4 C2 ;
```

Don't enter this yet. The 1 C will play low C for one beat, 4 C2 will play high C for four beats.

Step 2 Define the notes of the musical scale using the new defining word created in step 1.

```
240 NOTE C   213 NOTE D   190 NOTE E   179 NOTE F
160 NOTE G   142 NOTE A   127 NOTE B   106 NOTE C2
```

The word NOTE is our compiler extension, the number in front of NOTE sets the frequency (actually the period) and the letter or word following NOTE is the name of the note (dictionary entry) created. The word NOTE works much like CONSTANT. Both NOTE and CONSTANT create a dictionary entry and store the stack number imediately after the dictionary entry. The difference is that when a constant executes it pushes its value stored at creation time on the data stack, when a note executes it takes the top of the stack as the number of beats to play the note requested.

Step 1 This is where we extend the compiler to include the note defining capability. Here is the definition:

```
: NOTE CREATE , DOES> @ TONE 500 * 0 DO I DROP LOOP OFF ;
```

For such a powerful programing feature, it sure doesn't look like much does it?

The part before the word DOES> and after NOTE describes how to make notes. The part after DOES> and before ; tells how to play a note, or what is to happen when you execute a note type object.

First look at the creation part (before the DOES>). You know from the last section that CREATE makes a dictionary entry for the word which follows in the input stream. This time, however, CREATE is included in a definition so the dictionary entry will not be created until NOTE is executed. At this time the word following in the input stream will be the name of the musical note we want to create. The comma (,) stores the top stack number in the dictionary space following the created note name (ie in the pfa). This value will later be recovered and used to control the sound generator when the note is played.

The part following DOES> tells how to play the note. Recall that whenever a word defined by CREATE executes (and in this case NOTE because it includes CREATE), the parameter field address gets pushed on the data stack. The @ which follows DOES> fetches the value stored at this location (the value stored by , at creation time) and sounds a TONE . 500 * multiplies the number of beats requested by 500 and we then hang in a do nothing loop for that number of counts. The last word turns the sound generator off.

Now you know why step 3 was saved till the end. Here are the definitions again, this time presented in the correct order. Remember you must re-enter the definitions for ON , OFF , and TONE .

```
: NOTE CREATE , DOES> @ 500 * 0 DO I DROP LOOP OFF ;
240 NOTE C   213 NOTE D   190 NOTE E   179 NOTE F
160 NOTE G   142 NOTE A   127 NOTE B   106 NOTE C2
: SCALE 1 C 1 D 1 E 1 F 1 G 1 A 1 B 4 C2 ;
SCALE
```

Opps, forgot all the (ret) OK's but you are probably tired of seeing that at the end of each line anyway. For a more complete version of this music language see Saturn Softnews volume 1 number 4.

In summary, there are three aspects to consider when using CREATE . . . DOES> .

1. The process of creating a new defining word (compiler extension) which will create a new class of objects with the desired qualities (musical in the case of our example).

2. The action of using the new defining word to create some objects of this class (some musical notes in our example).

3. The execution of one or more of the new objects resulting in actions characteristic of their class. In our example some notes were included in the definition of the word SCALE so that when SCALE was executed that is what we heard.

This is really the most exciting aspect of FORTH. No longer do we computer users have to be content to live with programming tools that the language implementor thought we might find useful. When programming in FORTH it is possible to extend the compiler in any direction that is found desirable.

Here is how you can create a data type called VECTOR. After defining VECTOR we will calculate the work W according to the following:

W = F dot D (vector dot product) where:

F = [1,2,3] is the FORCE and

D = [4,6,5] is the DISTANCE.

Let' do it then:

```
: VECTOR CREATE 2* ALLOT DOES> SWAP 1- 2* + ; (ret) OK

3 VECTOR DISTANCE     3 VECTOR FORCE (ret) OK
```

Assign some values to our vectors:

```
4 1 DISTANCE !    6 2 DISTANCE !    5 3 DISTANCE ! OK
1 1 FORCE !       2 2 FORCE !       3 3 FORCE ! OK

: WORK 0 4 1 DO I DISTANCE @ I FORCE @ * + LOOP
            ." THE WORK IS " . ; OK
  WORK (ret) THE WORK IS 31 OK
```

NUMBER FORMATTING OPERATORS

Here is a summary of the number formatting operators available in FORTH. Check the corresponding glossary entries for a more complete description of each word.

<# Start numeric output string conversion.

Convert next digit of number and add character to the output strint.

#S Convert all remaining significant digits of unsigned double number to output string.

HOLD Add ASCII character on top of stack to the output string.

SIGN Add minus sign to output string if number is negative.

#> Terminate numeric conversion leaving address and count for the TYPE command.

To illustrate the use of the number formatting operators we present a complete program for your study and analysis. The program is heavily commented and you should carefully study each word and phrase until you understand their purpose. The number formatting operators are used in the word REAL.

```
Screen 10   A hex
 0 ( NUMBER INPUT WITH USER ERROR RECOVERY-0      J.W.B. 15:01:82 )
 1 -->
 2 ( INPUT - This word is used to input one positive or negative )
 3 (         single or double precision number from the console   )
 4 (         device.  The inputed number will be found on top of  )
 5 (         the data stack.  If an invalid number is given by    )
 6 (         the user the message "INVALID, REDO FROM START" will )
 7 (         be printed allowing another chance for correct entry.)
 8 (         The normal FORTH response would be to give a system  )
 9 (         error message and then ABORT the executing FORTH     )
10 (         program.                                             )
11 (         When control is returned to the calling program it   )
12 (         should check the variable DPL to determine if the    )
13 (         user inputed a single or double precision number.    )
14 (         A -1 indicates a single precision number.  any other )
15 (         value gives decimal location in a double number.     )

Screen 11   B hex
 0 ( NUMBER INPUT WITH USER ERROR RECOVERY-1      J.W.B. 15:01:82 )
 1 FORTH DEFINITIONS HEX
 2 : INPUT  BEGIN              ( MAIN LOOP TO ALLOW FOR RETRIES    )
 3            TIB @ 10 EXPECT ( INPUT 16 CHARACTERS               )
 4            0 >IN !         ( POINT TO START OF INPUT           )
 5            BL WORD         ( PARSE WORD AND MOVE TO HERE       )
 6            0 0 ROT         ( DOUBLE ZERO, ROT ADDRESS TO TOP   )
 7            DUP 1+          ( DUP ADDRESS, POINT TO SIGN        )
 8            C@  2D =        ( LEAVE TRUE FLAG IF -VE SIGN       )
 9            DUP             ( DUP THE SIGN FLAG, 1=-VE,0=+VE    )
10            >R  +           ( FLAG TO R-STACK, INC ADDRS BY FLAG)
11            -1              ( INITIAL VALUE FOR DPL             )
12            BEGIN           ( INNER LOOP TO DO CONVERSION       )
13              DPL !         ( SET DPL                           )
14              CONVERT       ( CONVERT TILL NON DIGIT            )
15 -->

Screen 12   C hex
 0 ( NUMBER INPUT WITH USER ERROR RECOVERY-2      J.W.B. 15:01:82 )
 1             DUP C@          ( GET CHAR THAT HALTED CONVERSION )
 2             2E =            ( WAS IT A PERIOD ?               )
 3             WHILE           ( IF SO SET DPL TO ZERO           )
 4             0               ( AND GO BACK TO                  )
 5             REPEAT          ( FINISH THE CONVERSION           )
 6           C@ BL -           ( WAS CONVERSION STOPED BY BLANK? )
 7           WHILE             ( IF NOT WE HAVE ERRONEOUS INPUT  )
 8           ." INVALID, REDO FROM START "
 9           CR                ( SO SEND MESSAGE AND             )
10           2DROP R> DROP     ( CLEAR STACKS FOR A              )
11           REPEAT            ( RETRY BY THE USER               )
12           R> IF DNEGATE THEN ( GET SIGN FLAG BACK AND NEGATE  )
13           DPL @ 0<          ( IF SINGLE NUMBER INPUT THEN     )
14           IF DROP THEN ;    ( CONVERT TO SINGLE NUMBER        )
15 DECIMAL -->
```



```
Screen 13   D hex
 0( PICTURED OUTPUT OF A REAL NUMBER-0            J.W.B. 15:01:82 )
 1 -->
 2
 3 ( REAL.  This word will allow the pictured formatting and      )
 4 (        output of a double precision number to include        )
 5 (        a decimal point located as specified by the value     )
 6 (        stored in the variable DPL . This word together       )
 7 (        with INPUT give previously will allow calculations    )
 8 (        with real numbers.  It is the responsibility of the   )
 9 (        programmer to handle the scaling of numbers and to    )
10 (        keep track of the decimal point.  programming your    )
11 (        calculations in this way has a definite speed         )
12 (        advantage over floating point calculations in such    )
13 (        applications as process control and data acquisition  )
14 (        where time is important.                              )
15

Screen 14   E hex
 0 ( PICTURED OUTPUT OF A REAL NUMBER-1           J.W.B. 15:01:82 )
 1 FORTH DEFINITIONS HEX
 2 : REAL.
 3       SWAP OVER       ( <sign> <number>                        )
 4       <#              ( set up for pictured output             )
 5       DPL @           ( fetch number of decimal places         )
 6       ?DUP            ( duplicate if non zero                  )
 7       IF              ( if non zero                            )
 8       0 DO            ( we assume that DPL >= 0                )
 9           #           ( convert DPL digits                     )
10         LOOP
11       2E HOLD         ( then ensert the decimal point          )
12       THEN  #S        ( now convert the remainder              )
13       SIGN #>         ( add the sign if necessary              )
14       TYPE SPACE ;    ( and type the number out.               )
15 -->

Screen 15   F hex
 0 ( EXAMPLE USING INPUT AND REAL.                J.W.B. 15:01:82 )
 1 FORTH DEFINITIONS DECIMAL
 2 : RADIUS?             ( input a valid radius from the user     )
 3       BEGIN           ( dont accept a negative radius          )
 4       CR ." ENTER THE RADIUS "
 5       INPUT           ( get number from user                   )
 6       DPL @ 0<        ( check for single number entered        )
 7       IF S->D 0 DPL ! THEN   ( and extend to double if so      )
 8       2DUP            ( duplicate so we have a copy to test    )
 9       DO< WHILE       ( check for -ve entry                    )
10       2DROP CR        ( drop the invalid number                )
11       ." A RADIUS IS POSITIVE!! "   ( let him know why         )
12       REPEAT ;        ( we leave radius on the stack as a      )
13                       ( double number and decimal places       )
14 -->                   ( stored in DPL                          )
15
```

```
Screen 16    10 hex
 0 ( EXAMPLE USING INPUT AND REAL.              J.W.B. 15:01:82 )
 1 : CIRCUMFERENCE       ( compute and print from double number )
 2       2DUP DROP       ( duplicate and treat as single number )
 3       2*              ( 2 x radius on stack                  )
 4       355 113 */      ( approximate pi by fraction           )
 5       S->D            ( extend to double number              )
 6       CR ." THE CIRCUMFERENCE IS "  ( crlf and message       )
 7       REAL.  ;        ( print answer formated with dec. pt.  )
 8 : AREA                ( compute from double radius on stack  )
 9       DROP DUP        ( treat as single number and duplicate )
10       *               ( radius squared on the stack          )
11       355 113 */ S->D  ( area extended to double number      )
12       DPL @ 2* DPL !   ( adjust number of decimal places     )
13       CR ." THE AREA IS "  ( crlf and message                )
14       REAL.  ;
15 : CIRCLE  RADIUS? CIRCUMFERENCE  AREA ;  ( This is it!!      )
```

RECURSIVE DEFINITIONS USING MYSELF

The following example uses the structure IF .... ELSE .... THEN and the word MYSELF to define a recursive power function. For the purposes of the discussion one asterisk "*" will be used for multiply and two asterisks "**" will be used for powers (** is not a FORTH word). That is to say, 2 cubed will be written 2**3. Here is the recursive raise to a power definition that will be used.

```
B**N  =  B * B ** (N-1)  if N is not zero

B**N  =  1               if N equals zero
```

```
POWER FUNCTION
        IF VALUE OF N>0 THEN CALL MYSELF WITH POWER N-1
                        AND MULTIPLY RESULT BY B
        ELSE WHEN N=0 ANSWER IS ALWAYS 1
        END
```

The above word description shows that the power function will continue to call itself until N is finally equal to 0. It will then unwind through the successive calls to itself and finally return with the answer. This repeated calling of itself is called recursion. Recursive functions are not easily written in BASIC. To implement recursive functions a stack structure is required.

Before the actual FORTH program is presented use the glossary to look up the words IF, ELSE, THEN, and MYSELF. Here is the screen which gives the recursive colon definition for the function POW.


```
Screen 17    11 hex
 0 ( USING RECURSION TO IMPLEMENT A POWER FUNCTION )
 1 ( USAGE    2 3 POW  .  (ret)  8  OK )
 2 FORTH DEFINITIONS          ( PUT IN FORTH VOCABULARY )
 3 : POW DUP                  ( DUPLICATE POWER N       )
 4       IF                   ( IF N NOT 0              )
 5          OVER SWAP         ( STACK NOW HAS B B N     )
 6          1 -               ( STACK NOW HAS B B N-1   )
 7          MYSELF *          ( CALL MYSELF AND MULT BY B )
 8       ELSE                 ( IF N=0                  )
 9          DROP DROP         ( DUMP B AND N            )
10          1                 ( REPLACE WITH 1          )
11       THEN ;               ( THAT'S ALL FOLKS        )
12
13
14
15
```

To compile the definition type 17 LOAD (ret) . Test the definition as shown in line 1 of the screen. Well, that worked just great, didn't it!! Here is a recursive definition of the factorial function for you to try.

```
N! = N * (N-1)!   if N is not equal to zero

N! = 1            if N is equal to zero
```

Here is the FORTH screen; you figure out how it works all by yourself!

```
Screen 18    12 hex
 0 ( RECURSIVE IMPLEMENTATION OF THE FACTORIAL FUNCTION )
 1 FORTH DEFINITIONS
 2 : FACT DUP          ( N N NOW ON STACK )
 3       IF DUP 1 -    ( IF N NOT ZERO N N-1 ON STACK )
 4          MYSELF *   ( CALL MYSELF AND MULTIPLY     )
 5       ELSE DROP 1   ( IF N=0 THEN ANSWER IS 1      )
 6       THEN ;
 7
 8
 9

 ....

15
```

THE 6502 FORTH ASSEMBLER

The 6502 FORTH assembler is automatically activated by the words CODE and ;CODE . The words CODE and ;CODE make the assembler the CONTEXT vocabulary, save the current system BASE on the stack and then set the system BASE to HEX. Later the word END-CODE restores the BASE to the former value. FORTH assembly language mnemonics should follow CODE and ;CODE . The words CODE and ;CODE are used as follows

1. To create a machine code word definition use the following form.

```
CODE <name> . . . assembly mnemonics . . . END-CODE
```

2. To create a new defining word with a machine language runtime routine.

```
: <name> . . words . . ;CODE . . assembly mnemonics . . END-CODE
```

The ;CODE plays the same role as DOES> . The part before ;CODE is high level FORTH and tells how to create a new object of the class <name> and the part after ;CODE is the machine language runtime routine used when an object of the class <name> is executed. The defining word <name> might be used in the same way as the defining words CONSTANT , VARIABLE and NOTE of our previous example.

ADDRESS MODE INDICATORS

All address mode indicators behave the same and simply store a one byte offset in the variable MVAR which later combines with a base value stored with each opcode mnemonic to produce a valid opcode. There are eleven valid address mode indicators. If an address mode indicator is missing then either absolute or zero page is assumed according to the operand address on the stack.

IM, Immediate address mode indicator.

ZP, Zero page address mode indicator.

AB, Absolute address mode indicator.

IX, Indirect X address mode indicator.

IY, Indirect Y address mode indicator.

AX, Absolute X address mode indicator.

AY, Absolute Y address mode indicator.

ZY, Zero page Y address mode indicator.

ZX, Zero page X address mode indicator.

X, Indexed by X assumming absolute or zero page automatically.

Y, Indexed by Y assumming absolute or zero page automatically.

RELATIVE BRANCH MNEMONICS

All relative branch mnemonics behave the same way. The appropriate relative branch opcode is retrieved from a field following the mnemonics dictionary entry and is compiled at HERE. Then the stack value is examined and if less than $FF is assumed to be a user determined offset and compiled following the relative branch opcode. If however the stack contains a value greater than $FF an offset is computed between the value of the stack and HERE+1 and compiled following the opcode instead. A check is also made for a relative branch out of range.

BCC, Branch if carry clear.

BCS, Branch if carry set.

BEQ, Branch if equal to zero.

BMI, Branch if minus.

BNE, Branch if not equal to zero.

BPL, Branch if plus.

BVC, Branch if overflow clear.

BVS, Branch if overflow set.

INSTRUCTIONS WITH ONE AND TWO BYTE OPERANDS

A base value following the dictionary entry of each mnemonic is retrieved and combined with the addressing mode offset to form a valid opcode which is then compiled into the next available dictionary location. Depending on the mnemonic type either one or two bytes of the operand which is now at top of the parameter stack is compiled following the opcode.

ADC, Add with carry.

AND, And with accumulator.

ASL, Shift memory left.

BIT, Bit test.

CMP, Compare with accumulator.

DEC, Decrement memory.

EOR, Exclusive or.

INC, Increment memory.

LDA, Load accumulator.

LSR, Shift memory right.

ORA, Inclusive or.

ROL, Rotate left.

ROR, Rotate right.

SBC, Subtract with borrow.

STA, Store accumulator.

STX, Store X.

STY, Store Y.

CPX, Compare with X.

CPY, Compare with Y.

LDX, Load X.

LDY, Load Y.

ASLA, Accumulator shift left.

LSRA, Accumulator shift right.

ROLA, Rotate accumulator left.

RORA, Rotate accumulator right.

ONE BYTE INSTRUCTIONS

The actual opcode which follows the dictionary entry for the mnemonic is retrieved and compiled into the next available dictionary location.

BRK, Break.

CLC, Clear carry.

CLD, Clear decimal mode.

CLI, Clear interrupt flag.

CLV, Clear overflow flag.

DEX, Decrement X.

DEY, Decrement Y.

INX, Increment X.

INY, Increment Y.

NOP, No operation.

PHA, Push accumulator to stack.

PHP, Push processor status.

PLA, Pull accumulator from stack.

PLP, Pull processor status.

RTI, Return from interrupt.

RTS, Return from subroutine.

SEC, Set carry.

SED, Set decimal mode.

SEI, Set interrupt.

TAX, Transfer accumulator to X.

TAY, Transfer accumulator to Y.

TSX, Transfer stack pointer to X.

TXA, Transfer X to accumulator.

TXS, Transfer X to stack pointer.

TYA, Transfer Y to accumulator.

JUMP INSTRUCTIONS

The jump instructions simply compile the opcode which follows the mnemonic's dictionary entry and the 16 bit address on the parameter stack.

JMP, Jump to address.

JMI, Indirect jump.

JSR, Jump to subroutine.

ASSEMBLER CONDITIONALS

The assembler conditionals which follow are based on the 6502 condition codes (N , Z , and C). They are used before the assembler macros IF, UNTIL, and WHILE to produce structured assembly

language programs.

0<, Result negative? or N=1?

0>, Result positive? or N=0?

0=, Result zero? or Z=1?

0<>, Result not zero? or Z=0?

CS, Carry set? or C=1?

CC, Carry clear? or C=0?

ASSEMBLER MACROS

The assembler macros BEGIN, UNTIL, IF, ELSE, THEN, WHILE, and REPEAT, are used in a manner similar to that in colon defintions. The assembler macros are used to control program flow at execution time. At assembly time, the macros generate the proper branching instructions and fixup the offsets so that the user does not have hand calculate offsets or use labels. See the corresponding entries in the FORTH vocabulary. The major difference is that IF, UNTIL, and WHILE, must be preceded by an assembler conditional. Below are examples of the allowed structures.

mnemonics BEGIN, mnemonics condition UNTIL, mnemonics

The machine instructions between BEGIN, and UNTIL, will be repeated until "condition" is true.

... condition IF, true mnemonics ELSE, false mnemonics THEN, ...

If the condition is true then machine code between IF, and ELSE, is executed and control is then passed to just after THEN, . If the condition is false then the machine code between ELSE, and THEN, is executed and control is then passed to just after THEN, .

mnemonics BEGIN, mnemonics condition WHILE, mnemonics REPEAT, ...

If the condition is true the machine code between WHILE, and REPEAT, will be executed, and then control will be passed back to just after BEGIN, . If the condition is false then control is passed to just after REPEAT, .

IMPORTANT ADDRESSES AND ENTRY POINTS

The addresses and entry points that follow have been defined as CONSTANTS and form part of the ASSEMBLER vocabulary. All CODE and ;CODE definitions will utilize these addresses and entry points to provide proper linkage with the FORTH system. Examples of their usage follows.

IP $00AE

Address of the interpretive pointer in zero page.

W $00B1

Address of the code field pointer in zero page.

N $00A6

Address of an 8 byte scratch area in zero page.

XSAVE $00B5

Address of temporary storage for parameter stack pointer.

YSAVE $00B6

Address for temporary storage for the Y register.

UP $00B3

Address of the user area pointer in zero page.

SETUP $033E

Address of routine to transfer data stack items to the 8 byte page zero scratch pad area labeled N . The number of data stack items to be transfered is put is the accumulator. A maximum of four stack items can be transfered. The transfered items are removed from the data stack. The top stack item will be found at N (lo) and N+1 (hi), the second from the top at N+2, N+3 etc.

PUT $025E

Address of routine to replace the present computation stack high byte from accumulator, and put from the machine stack one byte which replaces the present low stack byte; continue on to NEXT.

PUSH $025C

Address of routine to repeat PUT but creating a new bottom item on the computation stack.

PUSHOA $06A2

Address of routine to place the accumulator at the low stack byte, with the high byte zero. This routine creates a new stack item

POP $0433

Address of routine to remove one 16 bit item from the parameter stack.

POPTWO $0431

Address of routine to remove two 16 bit items from the parameter stack.

NEXT $0433

Address of the inner-interpreter, to which all code routines must return. NEXT fetches indirectly, referred to IP, the next compiled FORTH word address. It then jumps indirectly to pointed machine code.

USING THE 6502 FORTH ASSEMBLER

The 6502 FORTH assembler is used to compile mnemonics and operands which follow CODE and ;CODE . The assembler uses postfix notation (reverse Polish) as in FORTH itself. This means that in assembly language code the operand comes before the operator. Each equivalent of a line of assembly code has a symbolic or numeric operand, an address mode modifier (if required) and finally the opcode or instruction mnemonic.

The 6502 X register is the data stack pointer. You must save and restore the X register using the location XSAVE if your code will use the X register.

NEXT is the main re-entry point for code definitions. Re-entry at NEXT leaves the data stack intact. Other common re-entry points are POP (removes one data stack entry), POPTWO (removes two), PUSH (returns one data stack value) and PUSHOA. It will take careful study of the examples to thoughly understand the usage of these re-entry points.

To access the top of the data stack from a code defintion the following are equivalent:

```
RAE-ASSEMBLER          6502 POSTFIX FORTH ASSEMBLER
 LDA *$00,X        00  ZX,  LDA,  or  TOP    LDA,   (LOW BYTE)
 LDA *$01,X        01  ZX,  LDA,  or  TOP 1+ LDA,   (HI BYTE)
```

To access the second item on the data stack:

```
 LDA *$02,X        02  ZX,  LDA,  or  SEC    LDA,   (LOW BYTE)
 LDA *$03,X        03  ZX,  LDA,  or  SEC 1+ LDA,   (HI BYTE )
```

Accessing the data stack in this way does not remove the number from the data stack. If you access the top of the stack in this way and you also want to remove it you should return via POP . If you access the top and the second item and you want to remove them both you should return via POPTWO. To access three or four data stack entries you should use the subroutine SETUP which automatically removes them from the data stack and transfers them to the 8 byte

page zero scratch area at N .

ASSEMBLER EXAMPLES

1. Two CODE defintions equivalent to the MULTIBEEP colon definition.

This examle shows to ways to use the assembler sturctured programming macros. It will call the SYM-1 BEEP subroutine at $8972 and the NOBEEP subroutine at $899B. The programs are presented in screens 34 and 35 which follow. Once you have the programs working try changing the parameters passed to the subroutine or otherwise customize the definitions.

```
Screen 34     22 hex
  0 ( ASSEMBLER BEGIN, . . . UNTIL, . . . EXAMPLE  J.W.B. 22:01:82 )
  1 FORTH DEFINITIONS
  2 CODE BEEPER1     ( <count>  BEEPER1 -> ----     beep count times )
  3      XSAVE       STX,                        ( SAVE X REG          )
  4      YSAVE       STY,                        ( SAVE Y REG          )
  5      TOP         LDA,                        ( GET THE COUNT       )
  6    BEGIN,        PHA,                        ( SAVE COUNT ON STACK )
  7     8972         JSR,                        ( CALL TO SYM BEEP    )
  8     899B         JSR,                        ( CALL TO SYM NOBEEP  )
  9     PLA,         TAX,                        ( GET COUNT IN X REG  )
 10     DEX,         TXA,                        ( DECREMENT THE COUNT )
 11     0=,        UNTIL,                        ( ARE WE DONE YET?    )
 12    XSAVE         LDX,     YSAVE      LDY,   ( RECOVER X AND Y     )
 13     POP          JMP,     END-CODE          ( REMOVE 1 AND RETURN )
 14     10     BEEPER1                          ( TEST VALUE          )
 15

Screen 35     23 hex
  0 ( ASSEMBLER BEGIN, . . WHILE, . . REPEAT, EXAMPLE J.W.B. 01:82 )
  1 FORTH DEFINITIONS
  2 CODE BEEPER2       (  <count>  BEEPER2 -> ---- beep count times )
  3      XSAVE     STX,   YSAVE     STY,    ( SAVE X AND Y REG      )
  4        TOP     LDA,     TAX,     INX,    ( GET COUNT+1 IN X REG  )
  5      BEGIN,    DEX,                      ( X REG = COUNT AGAIN   )
  6        0<>,    WHILE,                    ( ARE WE DONE?          )
  7        TXA,    PHA,                      ( SAVE COUNT ON STACK   )
  8       8972     JSR,                      ( CALL SYM BEEP         )
  9       899B     JSR,                      ( CALL SYM NOBEEP       )
 10        PLA,    TAX,                      ( RECOVER THE COUNT     )
 11      REPEAT,                             ( BACK TO BEGIN,        )
 12      XSAVE     LDX,    YSAVE    LDY,    ( RESTORE X AND Y       )
 13       POP      JMP,    END-CODE         ( REMOVE THE COUNT      )
 14   10 BEEPER2                            ( TEST IT               )
 15
```

2. Real Time Clock for FORTH

This is a practical application of the assembler which illustrates the integration of high level FORTH with assembly

language CODE definitions. The programs are presented in screens 25, 26, 27, and 28 which follow. Below we explain some of the lines of code.

Screen 25

Line 2: JIFFY is a single precision constant used to count 20 "jiffys" to the second, it is initialized to -1. This is required so that the first time through (INTRPT) it will be reset to -20. SECS is a double precision variable (32 bits) which is initialized to 0. The FORTH clock will count only seconds and we will convert to the format HH:MM:SS when we output the time.

Line 3: (INTRPT) is the routine that executes when the VIA timer 1 times out. ' JIFFY pushes the PFA address of the constant JIFFY on the data stack. This location is incremented and compared to zero in line 4.

Lines 4 through 6: If JIFFY is zero as a result of the increment then the double precision variable SECS is incremented. Note the use of assembler conditionals and macros to check for the carry into successively higher bytes of SECS. Also note that SECS is a variable and does not require the "tick" to access the PFA as was required with the constant JIFFY.

Lines 7 through 14: Here we compare the double precision variable SECS to $15180, the number of seconds in one day base 16. If a match is obtained the variable SECS is zeroed. This occurs in lines 10, 11, and 12. Line 13 resets the constant JIFFY to -20. In line 14 the interrupt is cleared, the accumulator restored and we return control to FORTH.

Screen 26

Lines 2 through 7: The CODE definition INTIME (initialize time) sets the PFA of (INTRPT) in the SYM user IRQ vector at $A678. Note how the FORTH assembler digs out the high and low bytes of (INTRPT)'s PFA address in lines 3 and 4.

Lines 8 through 13: These lines configure the VIA timer #1 to generate continuous interrupts. Lines 10 and 11 set the speed at which the clock runs. (I think it runs a little slow, who will send me the right values?) In lines 12 and 13 we reset JIFFY to -20 and return to FORTH with a jump to NEXT. It should be noted that the routine INTIME can be executed from FORTH whereas the routine (INTRPT) would cause a system crash if executed from FORTH.

Screen 27 Formatting the time.

Lines 10 through 13: The sequence <# :00 :00 # # #> in the word FTIME is responsible for formatting the time as HH:MM:SS at PAD so that it can be output by the TYPE command.


```
Screen 25    19 hex
 0 ( REAL TIME  SK-FORTH79 CLOCK-1                    J.W.B. 09:12:81 )
 1 FORTH DEFINITIONS HEX
 2 -1 CONSTANT JIFFY   2VARIABLE SECS     0.0 SECS 2!
 3 CODE (INTRPT)       ( THIS IS THE INTERRUPT ROUTINE                 )
 4              PHA,  ' JIFFY     INC,  ( INCREMENT JIFFY              )
 5      0=, IF, SECS 2+  INC,  0=, IF, SECS 3 +     INC,
 6      0=, IF, SECS     INC,      THEN,      THEN,
 7              SECS          LDA,    01    IM, CMP,
 8      0=, IF, SECS 3 +      LDA,    51    IM, CMP,
 9      0=, IF, SECS 2+       LDA,    80    IM, CMP,
10      0=, IF, 0         IM, LDA,  SECS 2+     STA,
11              SECS 3 +      STA,  SECS        STA,
12        THEN,             THEN,             THEN,
13              EC        IM, LDA, ' JIFFY      STA,
14        THEN, A804            LDA,         PLA, RTI,   END-CODE
15 -->

Screen 26    1A hex
 0 ( REAL TIME  SK-FORTH79 CLOCK-2                    J.W.B. 09:12::81 )
 1
 2 CODE INTIME  ( ROUTINE TO SET UP 6522 AND SET INTERRUPT VECTOR )
 3        SEI,  ' (INTRPT)  ( DISABLE INTERUPTS, GET ADDRESS      )
 4    0   100    U/MOD     ( CONVERT ADDRESS TO HI AND LOW BYTE   )
 5    IM, LDA,  A679 STA,  ( SET HI  BYTE OF INTERRUPT ADDRESS    )
 6    IM, LDA,  A678 STA,  ( SET LOW BYTE OF INTERRUPT ADDRESS    )
 7
 8   C0 IM, LDA,    A80E  STA,  ( SET IER TO ENABLE T1 INTERRUPTS )
 9   40 IM, LDA,    A80B  STA,  ( SET ACR SO T1 GENERATES INT'S   )
10   42 IM, LDA,    A806  STA,   ( SET T1 LO ORDER LATCH          )
11   C3 IM, LDA,    A805  STA,   ( SET T1 HI ORDER COUNTER        )
12   EC IM, LDA,  ' JIFFY STA,  ( SET JIFFY TO -20              )
13         CLI,    NEXT  JMP,  ( ENABLE INTERRUPTS AND GO BACK  )
14      END-CODE
15 -->

Screen 27    1B hex
 0 ( REAL TIME  SK-FORTH79 CLOCK-3                    J.W.B. 09:12:81 )
 1 FORTH DEFINITIONS HEX
 2
 3 : :00  #                ( CONVERT DIGIT BASE 10)
 4        6  BASE !        ( SWITCH TO BASE 6     )
 5        #                ( CONVERT DIGIT BASE 6 )
 6        3A HOLD          ( INSERT COLON IN OUTPUT STRING )
 7        DECIMAL  ;       ( SWITCH BACK TO BASE 10 )
 8
 9 : FTIME                 ( PRINT TIME  AS  HH:MM:SS )
10       <# :00           ( CONVERT SECONDS      :SS )
11          :00           ( CONVERT MINUTES    :MM:SS )
12          # #           ( CONVERT HOURS    HH:MM:SS )
13                #>      ( END OF CONVERSION        )
14       TYPE SPACE ;     ( TYPE TIME AND A SPACE    )
15 -->
```

```
Screen 28    1C hex
 0 ( REAL TIME  SK-FORTH79 CLOCK-4                J.W.B. 09:12:81 )
 1 FORTH DEFINITIONS HEX
 2 : TIME                    ( PRINT TIME AT TERMINAL )
 3       SECS 2@             ( FETCH THE SECONDS COUNT)
 4       FTIME  ;            ( FORMAT AND PRINT IT    )
 5
 6 : STOP             ( STOP THE CLOCK BEFORE YOU FORGET IT )
 7       40 A80E C!          ( DISABLE INTERUPTS      )
 8       00 A80B C! ;        ( RESET CONTROL REGISTER )
 9 DECIMAL
10 : SET    ( <hh> <mm> <ss>  SET -> ---- SET TIME AND START )
11      SWAP ROT 60 *        ( CONVERT hh TO MINUTES    )
12      + 60 M*              ( ADD mm CONVERT TO SECONDS )
13      ROT 0 D+             ( TOT SECONDS IS DOUBLE NUMBER )
14      SECS 2!              ( STORE IN SECONDS COUNTER  )
15      INTIME ;             ( INITIALIZE INTERRUPT ROUTINE )

Screen 29    1D hex
 0 ( PASSING  PARAMETERS-METHOD ONE               J.W.B. 22:01:82 )
 1 FORTH DEFINITIONS          ( CREATION OF THE WORD  TONE        )
 2 ( <volume> <duration> <period>  TONE -> ----                   )
 3 ( We will pass the stack parameters to the MTU BEEP subroutine )
 4 ( THIS EXAMPLE WILL NOT EXECUTE ON A SYM OR KIM!!! SEE TEXT    )
 5 CODE  TONE
 6          03 IM, LDA,   ( Set up to transfer three parameters  )
 7        SETUP     JSR,    ( Move them to scratch pad at N       )
 8        XSAVE     STX,    ( Must always save X register here    )
 9            N     LDY,    ( top stack at N , N+1   =  <period>  )
10         N 2+     LDX,    ( top-1 is at  N+2 , N+3 =  <duration>)
11        N 4 +     LDA,    ( top-2 is at  N+4 , N+5 =  <volume>  )
12         038D     JSR,    ( Call MTU  BEEP subroutine.          )
13        XSAVE     LDX,    ( Restore the X registier             )
14        NEXT      JMP,    ( This is always the return point.    )
15

Screen 30    1E hex
 0 ( PASSING  PARAMETERS-METHOD TWO-1             J.W.B. 22:01:82 )
 1 FORTH DEFINITIONS          ( CREATION OF THE WORD  TONE        )
 2 ( <volume> <duration> <period>  TONE -> ----                   )
 3 ( We will pass the stack parameters to the MTU BEEP subroutine )
 4 ( THIS EXAMPLE WILL NOT EXECUTE ON A SYM OR KIM!! SEE TEXT     )
 5 CODE  TONE
 6       XSAVE     STX,    ( Must always save X register here    )
 7       SEC 2+    LDA,    ( Get <volume> third from the top     )
 8                 PHA,    ( Save on stack                       )
 9       TOP       LDA,    ( Get <period> on top of the stack    )
10                 TAY,    ( Volume must be in Y register        )
11       SEC       LDA,    ( Get <duration> second from top      )
12                 TAX,    ( Duration must be in X register      )
13                 PLA,    ( Recover the volume saved on stack   )
14       038D      JSR,    ( Call MTU BEEP subroutine            )
15 -->
```



```
Screen 31    1F hex
 0 ( PASSING  PARAMETERS-METHOD TWO-2             J.W.B. 22:01:82 )
 1 ( CODE DEFINITON OF TONE CONTINUED                             )
 2
 3       XSAVE     LDX,       ( Restore data stack pointer       )
 4                 INX,       ( Pop one parameter off stack      )
 5                 INX,
 6       POPTWO    JMP,       ( Pop two more and return this way )
 7       END-CODE
 8
 9 ( TEST VALUES )
10
11 200 100 10 TONE     100 100 10 TONE
12 200 100 10 TONE     200  50 10 TONE
13 200 100 10 TONE     100 100 90 TONE
14
15

Screen 32    20 hex
 0 ( CREATION OF CONSTANT WITH ;CODE              J.W.B. 22:01:82 )
 1
 2 FORTH DEFINITIONS
 3
 4 : CONSTANT   CREATE             ( CREATE DICTIONARY ENTRY      )
 5              ,                  ( STORE DEFINED VALUE AT PFA   )
 6              ;CODE              ( DEFINE RUNTIME ROUTINE       )
 7              2      IM,  LDY,   ( POINT TO PFA WITH CFA        )
 8              W      IY,  LDA,   ( GET LOW DATA BYTE            )
 9                          PHA,   ( PUSH ON MACHINE STACK        )
10                          INY,   ( POINT TO THE HIGH BYTE       )
11              W      IY,  LDA,   ( FETCH THE HIGH BYTE          )
12              PUSH        JMP,   ( PUSH TO DATA STACK AND RETURN)
13              END-CODE
14
15

Screen 33    21 hex
 0 ( CREATION A NEW DEFINING WORD WITH ;CODE      J.W.B. 22:01:82 )
 1
 2 ( THE NEW WORD " BYTE " WORKS LIKE CONSTANT EXCEPT THAT ITS    )
 3 ( ITS RANGE IS 0 - 255 AND IT ONLY TAKES ONE BYTE FOLLOWING PFA)
 4
 5 FORTH DEFINITIONS
 6
 7 : BYTE       CREATE             ( CREATE DICTIONARY ENTRY      )
 8              C,                 ( STORE DEFINED VALUE AT PFA   )
 9              ;CODE              ( DEFINE RUNTIME ROUTINE       )
10              2      IM,  LDY,   ( POINT TO PFA WITH CFA        )
11              W      IY,  LDA,   ( GET LOW DATA BYTE            )
12              PUSHOA      JMP,   ( PUSH TO DATA STACK AND RETURN)
13              END-CODE
14
15
```

Lines 3 through 7: The word :00 converts one digit base 10, one digit base 6 and then inserts a colon, ":", in the output stream. It is most amazing that you can switch the system base mid way through the conversion of a number that you are preparing for output. But with FORTH anything is possible!!!

Screen 28 The useful words.

Lines 2 through 4: Use the word TIME to print the time on the console.

Lines 6 through 8: Use the word STOP the clock before forgetting it.

Lines 10 through 15: The routine set is used to start the clock and set the current time. If you are still reading, you should be able to figure out how it does this youself.

3. A Three Parameter TONE Generator.

This example was used in our doucumentation of MTU-FORTH79 and although we do not have the same subroutine on our KIM/SYM systems the example was retained for its educational value in that it demonstrates two different methods of passing paramters from FORTH's data stack to a CODE definition. The MTU-BEEP subroutine that is refered to requires three parameters. The volume (0 - FF) in the accumulator, the period in the Y register, and the duration in the X register. Clever harware types, with the MTU-DAC could route their CB2 sound through the onboard amplifier and then write a similar subroutine. Note again that this example will NOT work on you SYM or KIM.

In this example we will pass three parameters from the data stack to our CODE definition TONE. The usage of TONE will be as follows:

<volume> <duration> <period> TONE -> ----

where the parameters are the those described above.

This time there are three data stack numbers to be passed to our CODE definition so we will use the subroutine SETUP to remove them from the data stack and transfer them to the page zero scratch pad area at N. After the call to SETUP the parameters will be found as follows:

Top of stack:	<period>	at	N (lo)	N+1 (hi)
Second from top:	<duration>	at	N+2	N+3
Third from top:	<volume>	at	N+4	N+5

The CODE definition for TONE using SETUP to pass the parameters will be found in screen 29. For comparison we present the same thing in screens 30 and 31 passing the parameters without using SETUP.

ANATOMY OF A DICTIONARY ENTRY

```
================================================================
! LN ! NAME FIELD ! LINK FIELD ! CODE FIELD ! PARAMETER FIELD ... !
================================================================
/\             /\            /\           /\
NFA            LFA           CFA          PFA
```

Name Field

Byte 1 Length Byte.
- Bits 0 - 4 Name length 0 - 31
- Bit 5 SMUDGE bit
- Bit 6 Precedence bit (1=IMMEDIATE word, 0=regular)
- Bit 7 Always 1 to indicate start of name field

Byte 2 ASCII characters for name start in this byte, and continue

Byte last Last letter of name with bit 7 high to indicate the end of the name.

Link Field

The link field contains the name field address (NFA) of the previous dictionary entry.

Code Field

The code field contains the address of the actual machine code to be used for this dictionary entry. For example, a colon definition would have the address of DOCOL, the machine code which interprets the addresses found in the parameter field. A constant's code field address would contain the address of DOCON, the machine code runtime routine for contants. A CODE definition's code field address would contain the routine's parameter field address (PFA), as this is where the FORTH ASSEMBLER deposits the machine code compiled from the mnemonics which follow CODE.

Parameter Field

The parameter field contains other word addresses in the case of a colon definition, data in the case of a constant definition, and machine code in the case of a CODE definition.

DICTIONARY ENTRY FOR A CONSTANT

As an example the dictionary entry of the CONSTANT LIMIT is presented in detail using RAE assembler format.

```
NFA--->LIMIT.HD   .BY   $85       ; NAME FIELD (length byte)
                  .BY   'LIMI'    ; ASCII characters for name
                  .BY   $D4       ; last name character bit 7 high
LFA--->           .SI   FIRST.HD  ; LINK FIELD
CFA--->LIMIT      .SI   DOCON     ; CODE FIELD
PFA--->           .SI   UAREA     ; PARAMETER FIELD (data since
                                  ; UAREA is value of LIMIT)
```

The code field address (CFA) points to DOCON the machine language run-time routine for the CONSTANT data type. When LIMIT is executed the run-time routine DOCON gets the 16 bit number stored in the parameter field (the value of UAREA in this case) and pushes it onto the parameter stack. Below is the machine code in RAE assembler format of the routine DOCON. It is very important to understand exactly what is going on if you hope to define new data types of your own. If at first you cannot follow what is going on read this section again later.

```
DOCON   LDY   #$02      ; POINT TO PFA WITH CFA
        LDA   (W),Y     ; GET LO DATA BYTE
                        ; W CONTAINS CFA
        PHA             ; PUSH ON MACHINE STACK
        INY             ; PREPARE TO GET HI BYTE
        LDA   (W),Y     ; GET HI DATA BYTE
        JMP   PUSH      ; PUSH NUMBER TO PARAMETER STACK
                        ; AND RETURN
```

Now go on to the next section if you understand the above example.

CREATING A NEW DATA TYPE USING ;CODE

Suppose that we have need for a new data type called BYTE. The new data type is going to work just like CONSTANT, except that we are only going to store one byte of data in the parameter field. The range for our new data type will, of course, be only 0 - 255. If we have need for constants in this range, using the data type BYTE will result in a space saving. If you look on screen 32 you will see how the defining word CONSTANT would be coded in FORTH using ;CODE . Screen 32 is on page 63 of the manual.

It might be a good idea to go back and review the section on constants before you go on. The high level FORTH code between CONSTANT and ;CODE creates the dictionary entry for the constant name and stores the defined value in the parameter field. The machine code between ;CODE and END-CODE is the run-time routine for the constant data type and is identical to DOCON on the previous page.

IMPORTANT! When we define a new constant only the portion between CONSTANT and ;CODE is executed. When we use the new constant the portion between ;CODE and END-CODE is executed.

To create our new data type BYTE we must make a change to the definition routine. We must change the (,) to (C,). We store a single byte in the parameter field rather than a word. We must also change the run-time routine between ;CODE and END-CODE so than only one byte is fetched from the parameter field when a constant is executed. The FORTH definition of the defining word BYTE is presented in screen 33. You will find screen 33 on page 63 of this manual. To test the definition type and observe the following:

```
33 LOAD (ret) OK
44 BYTE XYZ (ret) OK
XYZ .  (ret) 44 OK
```


INTRODUCTION TO GLOSSARIES

Order

These glossaries contain all the word definitions in release 2.0 of SK-FORTH79. They are presented in alphabetical order using the ASCII sequence. Shown below, for reference, is the order of the ASCII symbols.

```
! " # $ % & ' ( ) * + , - . / 0...9 : ; < = > ? @ A...Z [ ]
```

Stack Notation

The second line of each entry shows a symbolic description of the action of the word on the parameter stack in terms of stack inputs and stack outputs. The top of the stack is to the right of both the stack inputs and outputs. Four dashes ---- indicate that either no inputs are required or that any inputs that were present have been removed. See the section entitled STACK MANIPULATION for a further explanation of the notation used.

The symbols used include:

addr	Memory address 0...65,535.
byte	8 bit byte (hi 8 bits zero) 0...255.
char	7 bit ASCII character (hi 9 bits zero) 0...127.
d	32 bit signed double integer, most significant portion with sign on top of the stack -2,147,483,648 ...2,147,483,647.
flag	Boolean flag (0=false, 1=true).
ff	Boolean false flag (ff=0).
n	16 bit signed integer number -32,768...32,767.
tf	Boolean true flag (tf=1).
u	16 bit unsigned integer 0...65,535.
ud	32 bit unsigned integer 0...4,294,967,296.

Input Text

<text>

Angle brackets will enclose arbitrary FORTH text to be accepted from the input stream. This notation refers to text from the input stream, not to values on the data stack. If the input stream is exhausted before encountering <text>, then an error condition exists.

Attributes

The capital letters in brackets following the word give its characteristics:

C	May only be used within a colon definition. A digit indicates the number of 16 bit memory words used if other than 1.
E	Run time code not intended for direct execution.
79	Definition required by the FORTH-79 Standard.
I	Has precedence bit set (will execute even when compiling). Also referred to as an IMMEDIATE word.
U	A user variable.
F	Word retained from the Fig FORTH Model.
S	Word added to SK-FORTH79 for its utility value or as part of the cassette or disk interface.

Stack Parameters

Unless otherwise noted, all references to numbers are for 16 bit signed integers. The high byte of a number is on top of the stack, with the sign in the leftmost bit. For 32 bit signed double numbers, the most significant part (with the sign) is on top of the stack.

All arithmetic is implicitly 16 bit signed integer math, with error and under-flow indication unspecified.

FORTH-79 Standard Application Programs

A standard application program may reference only the definitions of the required FORTH-79 word set (those designated with the attribute 79 following the glossary entry). If your program requires other definitions, ie., those designated with an F or M in the glossary entry, then they must be recoded and included with your application program. The glossaries include the FORTH code for all colon definitions used in SK-FORTH79.

We recommend that serious application programmers obtain and study the document:

The FORTH-79 Standard - available from
FORTH Interest Group P.O. Box 1105 San Carlos CA 94070

$00 or NULL (F)

This one character word consisting of ASCII null is the execution procedure used to terminate interpretation of a line of text from the terminal or within a disk buffer. Both the terminal and disk buffers always have at least one null at the end.

```
: 'null'  BLK @ IF 1 BLK +! 0 >IN ! BLK @ 0 B/SCR U/MOD
          DROP 0= IF ?EXEC R> DROP THEN ELSE R> DROP
          THEN ;
```

! (79)

n addr ! -> ----

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

```
CODE ! assemblby mnemonics END-CODE
```

!CSP (F)

---- !CSP -> ----

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

```
: !CSP SP@ CSP ! ;
```

 # (79)

d1 # -> d2

Generates 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 also #S.

```
: # BASE @ M/MOD ROT 9 OVER < IF 7 + THEN 30 + HOLD ;
```

#> (79)

d #> -> addr count

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

```
: #> DROP DROP HLD @ PAD OVER - ;
```

#S (79)

d1 #S -> d2

Generates ASCII text in the text output buffer, by the use of #, until a zero double number d2 results. Used between <# and #>.

```
: #S  BEGIN # OVER OVER OR 0= UNITL ;
```

' **(I,79)**

Used in the form:

```
----    ' <name>    ->    addr
```

Leaves the parameter field address (PFA) of the dictionary word <name>. As a compiler directive, executes in a colon definition to compile the address as a literal. If the word <name> is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. Within a colon-definition ' <name> is idenitcal to [' <name>] LITERAL. Pronounced "tick".

```
: '  -FIND 0= 0 ?ERROR DROP [COMPILE] LITERAL ; IMMEDIATE
```

(**(I,79)**

Used in the form: (ccccccc)

Ignores a comment that will be delimited by a right parenthesis on the same line. May occur during execution of a colon definition. A blank after the leading parenthesis is required. An error condition exists if the input stream is exhausted before the right parenthesis.

```
: (  29 WORD DROP ; IMMEDIATE
```

(ABORT) **(E,F)**

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

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

(.") **(C+,E,F)**

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

```
: (.")  R@ COUNT DUP 1+ R> + >R TYPE ;
```

(;CODE) **(C,E)**

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


```
: (;CODE)  R> LATEST PFA CFA ! ;
```

(+LOOP) **(C2,E,F)**

```
n    (+LOOP)    ->    ----
```

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

```
CODE  (+LOOP)  assembly mnemonics  END-CODE
```

(DO) **(C,E,F)**

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

```
CODE  (DO)  assembly mnemonics  END-CODE
```

(FIND) **(E,F)**

```
addr1    addr2    (FIND)    ->    pfa byte  tf    (match )
addr1    addr2    (FIND)    ->    ff              (no match )
```

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

```
CODE  (FIND)  assembly mnemonics END-CODE ;
```

(LINE) **(F)**

```
n1    n2    (LINE)    ->    addr    count
```

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

```
: (LINE)  >R C/L B/BUF */MOD R> B/SCR * +
          BLOCK + C/L ;
```

(LOOP) **(C2,E,F)**

The run-time procedure compiled by LOOP which increments the loop index by 1 and tests for loop completion. See LOOP.

```
CODE  (LOOP)  assembly mnemonics END-CODE
```

(MDISK) **(S)**

Call to the main HDE FODS operating system subroutine. This routine is not intended for user execution. Usage of this routine assumes that the requested command and parameters have first been set in the FODS parameter passing area at

$7200-$7220 for SYM and $F200-$F220 for KIM. For an example of its usage see the disk copy routine in the appendix.

CODE (MDISK) assembly mnemonics END-CODE

(NUMBER) (F)

d1 addr1 (NUMBER) -> d2 addr2

Converts the ASCII text beginning at addr1+1 with respect to BASE. The new value is accumulated into a double number d1, being left as d2. addr2 is the address of the first unconvertable digit. (NUMBER) is called CONVERT in the 79-Standard. Used by NUMBER.

```
: (NUMBER)  BEGIN 1+ DUP >R C@ BASE @ DIGIT
            WHILE SWAP BASE @ U* DROP ROT BASE @ U*
                  D+ " @ 1+ IF 1 DPL +! THEN R>
            REPEAT R> ;
```

(ROLL) (E,S)

n (ROLL) -> ----

Run time routine for ROLL. Equivalent to ROLL except that a check is not made for the stac5 out of bounds.

CODE (ROLL) assembly mnemonics END-CODE

(SET) (S)

addr1 addr2 blk flag (SET) -> ----

This word is called by the main disk read/write linkage to set the command, addresss, and track/sector information in the FODS paramater passing area. This routine is not intended for user execution. The addrsses in the definition below are for the SYM version. addr1 is low memory address, addr2 is hi memory address, blk is the block number being transfered and flag is true for read and false for a write operation.

```
: (SET)  0= 7218 C! 10 /MOD 7216 C! 1+ 7217 C! DUP
         B/BUF 1- + 7212 ! 7210 ! MDISK OFF ;
```

***** (79)

n1 n2 * -> product

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

```
: * U* DROP ;
```

***/** (79)

n1 n2 n3 */ -> n4

Leaves 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 SWAP DROP ;
```

***/MOD** (79)

n1 n2 n3 */MOD -> n4 n5

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

```
: */MOD  >R M* R> M/ ;
```

+ (79)

n1 n2 + -> sum

Leaves the sum of n1 and n2.

CODE + assembly mnemonics END-CODE

+! (79)

n addr +! -> ----

Adds n to the word value at address addr. Pronounced "plus-store".

CODE +! assembly mnemonics END-CODE

+- (F)

n1 n2 +- -> n3

Applies the sign of n2 to n2 which is left on the data stack as n2.

```
: +-  0< IF NEGATE THEN ;
```

+BUF (F)

addr1 +BUF -> addr2 f

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

```
: +BUF  B/BUF 4 + + DUP LIMIT =
        IF DROP FIRST THEN DUP PREV @ - ;
```

+LOOP (I,C2,79)

```
n1    +LOOP   ->   ----          (at runtime)
0ddr   n2     +LOOP   ->   ----  (at compile time)
```

Used in a colon defintion in the form:

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

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

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

```
: +LOOP  3 ?PAIRS COMPILE (+LOOP) BACK ; IMMEDIATE
```

+ORIGIN (F)

```
n   +ORIGIN   ->   addr
```

Leaves the memory address relative by n to the origin parameter area. n is the number of bytes offset from ORIGIN ($0200 for SYM, $2000 for KIM). This definition is used to access or modify the boot-up parameters and vectors in this area.

```
: +ORIGIN  200 + ;
```

, (79)

```
n   ,   ->   ----
```

Stores n into the next available dictionary memory cell, advancing the dictionary pointer. Pronounced "comma".

```
: , HERE ! 2 ALLOT ;
```

- (79)

```
n1   n2   -   ->   difference
```

Leaves the difference of n1-n2.

```
: - NEGATE + ;
```


--> (I,F)

Continues interpretation with the next disk screen. Pronounced "next-screen".

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

-FIND (F)

```
----   -FIND   ->   pfa  byte   tf    (if found)
----   -FIND   ->   ff               (if not found)
```

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

```
: -FIND  BL WORD CONTEXT @ @ (FIND) DUP 0=
         IF DROP HERE LATEST (FIND) THEN ;
```

-TRAILING (79)

```
addr   n1   -TRAILING   ->   addr   n2
```

Adjusts the character count n1 of a text string beginning at address to suppress output of trailing characters less than or equal to blank ($20). That is, the characters at addr+n2 to addr+n1 are blanks or control characters.

```
: -TRAILING  DUP 0 DO OVER OVER + 1 - C@ BL >
             IF LEAVE ELSE 1 - THEN LOOP ;
```

. (79)

```
n   .   ->   ----
```

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

```
: . S->D D. ;
```

." (I,79)

Used in the form: ." ccccccc"

Compiles an in-line string ccccc (delimited by 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 " .

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

.LINE (F)

line scr .LINE -> ----

Prints on the terminal device a line of text from the disk by its line and screen numbers. Trailing blanks are suppressed.

```
: .LINE (LINE) -TRAILING TYPE ;
```

.R (F)

n1 n2 .R -> ----

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

```
: .R >R S->D R> D.R ;
```

/ (79)

n1 n2 / -> quotient

Leaves the signed quotient of n1/n2.

```
: / /MOD SWAP DROP ;
```

/MOD (79)

n1 n2 /MOD -> remainder quotient

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

```
: /MOD >R S->D R> M/ ;
```

0 1 2 3 (F)

---- 0 1 2 or 3 -> 0 1 2 or 3

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

0< (79)

n 0< -> f

Leaves a true flag if the number is less than zero (negative); otherwise leaves a false flag.

CODE 0< assembly mnemonics END-CODE

0= (79)

n 0= -> f

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

CODE 0= assembly mnemonics END-CODE

0> (79)

n 0> -> f

Leaves a true flag if the number is greater than zero (positive); otherwise leaves a false flag.

CODE 0> assembly mnemonics END-CODE

OBRANCH (C2,E,F)

f OBRANCH -> ----

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.

CODE OBRANCH assembly mnemonics END-CODE

1+ (79)

n1 1+ -> n2

Increments n1 by 1 to give n2.

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

1- (79)

n1 2- -> n2

Decremenu n1 by 1 to give n2

```
: 1- 1 - ;
```

2! (79)

d addr 2! -> ----

Store double number d in 4 consecutive bytes beginning at addr, as for a double number.

```
: 2! SWAP OVER ! 2+ ! ;
```

2* (S)

n1 2* -> n2

Multiply the top number on the data stack n1 by 2 and leave as n2.

```
: 2* DUP + ;
```

2+ (79)

n1 2+ -> n2

Increments n1 by 2 to give n2.

```
: 2+ 2 + ;
```

2- (79)

n1 2- -> n2

Decrement n1 by 2 to give n2.

```
: 2- 2 - ;
```

2@ (79)

addr 2@ -> d

Leave on the stack the contents of the four consecutive bytes beginning at addr, as for a double number.

```
: 2@ DUP 2+ @ SWAP @ ;
```

2CONSTANT (79)

d 2CONSTANT <name> -> ----

A defining word used as shown above to create a dictionary entry for <name>, leaving d in its parameter field. When <name> is later executed, d will be left on the stack.

```
: 2CONSTANT CREATE , , DOES> 2@ ;
```

2DROP (79)

d 2DROP -> ----

Drop the top double number on the data stack.

```
: 2DROP DROP DROP ;
```


2DUP (79)

d 2DUP -> d d

Duplicate the top double number on the data stack.

```
: 2DUP OVER OVER ;
```

2OVER (79)

d1 d2 2OVER -> d1 d2 d1

Leave a copy of the second double number on the stack.

```
: 2OVER 4 PICK 4 PICK ;
```

2ROT (79)

d1 d2 d3 2ROT -> d2 d3 d1

Rotate the third double number to the top of the data stack.

```
: 2ROT 6 ROLL 6 ROLL ;
```

2SWAP (79)

d1 d2 2SWAP -> d2 d1

Exchange the top two double numbers on the data stack.

```
: 2SWAP ROT >R ROT R> ;
```

2VARIABLE (79)

---- 2VARIABLE <name> -> ----

A defining word used as shown above to create a dictionary entry of <name> and assign 4 bytes for storage in the parameter field. When <name> is later executed, it will leave the address of the first byte of its parameter field on the stack. All variables are initialized to zero.

```
: 2VARIABLE CREATE 0 0 , , DOES> ;
```

3* (S)

n1 3* -> n2

Multiply the stack number n1 by 3 and leave as n2.

```
: 3* DUP DUP + + ;
```

79-STANDARD (79)

Execute assuring that a FORTH-79 Standard system is available, otherwise an error condition exists.

```
: 79-STANTARD ;
```

: (I,C,79)

Used in the form called a colon definition:

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

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

```
: : ?EXEC !CSP CURRENT @ CONTEXT ! CREATE SMUDGE
    ] (;CODE) assembly mnemonics END-CODE
    IMMEDIATE
```

; (I,C,79)

Terminates a colon definition and stops further compilation. Compiles the run-time ;S .

```
: ; ?CSP COMPILE ;S SMUDGE [COMPILE] [ ; IMMEDIATE
```

;CODE (I,C,F)

Used in the form:

```
: <name> .... ;CODE assembler mnemonics END-CODE
```

Stops compilation and terminates a new defining word <name> by compiling (;CODE). Sets the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following assembler mnemonics up to END-CODE . When <name> later excutes in the form

```
<name> <name1>
```

the word <name> will be created with its execution procedure given by the machine code following name. That is, when <name1> is executed, it does so by jumping to the code after <name>. An existing defining word must exist in <name> prior to the ;CODE .

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


;S (I,F)

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

```
CODE ;S assembly mnemonics END-CODE
```

< (79)

```
n1 n2 < -> flag
```

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

```
CODE < assembly mnemonics END-CODE
```

<# (79)

Sets up for pictured numeric output formating using the words:

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

The conversion is done on a double number producing text at PAD in the form of an ASCII string.

```
: <# PAD HLD ! ;
```

= (79)

```
n1 n2 = -> flag
```

Leaves a true flag if n1=n2, otherwise a false flag.

```
: = - 0= ;
```

> (79)

```
n1 n2 > -> flag
```

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

```
: > SWAP < ;
```

>IN (U,79)

```
---- >IN -> addr
```

Leave the address of a variable which contains the present character offset within the input stream (0...1024). Used in WORD (and FIND.

>R (79)

n >R -> ----

Transfer n to the return stack. Every >R must be balanced by a R> in the same control structure nesting level of a colon definition.

```
CODE >R assembly mnemonics END-CODE
```

? (79)

addr ? -> ----

Prints the value contained at the address in free format according to the current value of BASE.

```
: ? @ . ;
```

?COMP (F)

Issues error message if not compiling.

```
: ?COMP STATE @ 0= 11 ?ERROR ;
```

?CSP (F)

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

```
: ?CSP SP@ CSP @ - 14 ?ERROR ;
```

?DUP (79)

n ?DUP -> n (n)

Duplicate n if it is non-zero. Pronounced "query-dup".

```
: ?DUP DUP IF DUP THEN ;
```

?ERROR (F)

flag n ?ERROR -> ----

Issues an error message number n, if the boolean flag is true.

```
: ?ERROR SWAP IF ERROR ELSE DROP ;
```

?EXEC (F)

Issues an error message if not executing.


```
: ?EXEC STATE @ 12 ?ERROR ;
```

?LOADING (F)

Issues an error message if not loading.

```
: ?LOADING BLK @ 0= 16 ?ERROR ;
```

?PAIRS (F)

n1 n2 ?PAIRS -> ----

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

```
: ?PAIRS - 13 ?ERROR ;
```

?STACK (F)

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

```
: ?STACK 8E SP@ U< 1 ?ERROR SP@ 10 U< 7 ?ERROR ;
```

?STK (F)

n ?STK -> n flag

Leave a true flag if n indexes an out of bounds data stack value otherwise leave a false flag. Used by PICK and ROLL for error detection.

```
: ?STK DUP DUP 0= SWAP DUP + SP@ + 88 > OR ;
```

?TERMINAL (F)

---- ?TERMINAL -> flag

Performs a test of the terminal break key. If the break key is hit followed by RETURN then the flag f is set to boolean true. If the break key is hit followed by SPACE then the flag f is set to boolean false. In both the previous cases execution is suspended until the key is hit. If the break key has not been hit then the flag f is set to boolean false

```
CODE ?TERMINAL asssembly mnemonics END-CODE
```

@ (79)

addr @ -> n

Leaves the 16 bit word contents of address.

```
CODE @ assembly mnemonics END-CODE
```

ABORT (79)

Clears the stacks and enter the execution state. Returns control to the operator's terminal, printing installation message.

```
: ABORT SP! DECIMAL CR ." SK-FORTH79 BY JOHN W. BROWN"
        CR CR ." COPYRIGHT (C) 1981 SATURN SOFTWARE
        LIMITED" [COMPILE] FORTH DEFINITIONS CR
        CR ." OK" QUIT ;
```

ABS (79)

```
n ABS -> u
```

Leaves the absolute value of n as u.

```
: ABS DUP +- ;
```

AGAIN (I,C2,F)

```
addr n AGAIN -> ---- (compiling)
```

Used in a colon definition in the form:

```
.... BEGIN .... AGAIN
```

At run-time, AGAIN forces execution to return to the corresponding BEGIN There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below).
At compile time, AGAIN compiled BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

```
: AGAIN 1 ?PAIRS COMPILE BRANCH BACK ; IMMEDIATE
```

AINIT (E,S)

This is the call to the ASSEMBLER used by CODE and ;CODE. This call holds the current system base on the stack, sets the BASE to HEX, initializes default zero page addressing and makes ASSEMBLER the context vocabulary. This call is not intended for user execution.

```
: AINIT ASSEMBLER BASE @ HEX SMVAR ;
```

ALLOT (79)

```
n ALLOT -> ----
```

Adds the signed number n to the dictionary pointer DP. May be used to reserve dictionary space or to re-origin memory.


```
: ALLOT DP +! ;
```

AND (79)

```
n1 n2 AND -> n3
```

Leaves the bitwise logical AND of n1 and n2 as n3.

```
CODE AND assembly mnemonics END-CODE
```

ASSEMBLER (F)

The name of the FORTH assembler vocabulary. Execution makes ASSEMBLER the CONTEXT vocabulary. CONTEXT is automatically set to ASSEMBLER by ;CODE and CODE.

B/BUF (F)

```
---- B/BUF -> n
```

A constant that leaves the number of bytes per disk buffer. The byte count is read from disk by BLOCK.

B/SCR (F)

```
---- B/SCR -> n
```

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

BACK (F)

```
addr BACK -> ----
```

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

```
: BACK HERE - , ;
```

BASE (U,79)

```
---- BASE -> addr
```

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

BEEP (S)

Generate a beep sound to attract humans attention.

```
CODE BEEP assembly mnemonics END-CODE
```

BEGIN (I,79)

---- BEGIN -> addr n (compiling)

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

```
: BEGIN ?COMP HERE 1 ; IMMEDIATE
```

BL (F)

---- BL -> char

A constant that leaves the ASCII value for "blank", namely $20.

BLANKS (F)

addr count BLANKS -> ----

Fills an area of memory beginning at addr with count blanks.

BLK (U,79)

----BLK -> addr

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

BLOCK (79)

n BLOCK -> addr

Leaves 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 the block n is read into the buffer. See also BUFFER, R/W, UPDATE, and SAVE-BUFFERS.

```
: BLOCK >R PREV @ DUP @ R@ - DUP + IF BEGIN +BUF
        0= IF DROP R@ BUFFER DUP R@ 1 R/W 2 - THEN
        DUP @ R@ - DUP + 0= UNTIL DUP PREV ! THEN
```



```
R> DROP 2+ ;
```

BOOT (S)

---- BOOT -> ----

A utility word used to initialize FORTH's virtual memory disk buffers and set WARNING for text error messages from screens 1 and 2.

```
: BOOT FIRST USE ! FIRST PREV ! EMPTY-BUFFERS
       1 WARNING ! ;
```

BRANCH (F,C2)

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.

```
CODE BRANCH assembly mnemonics END-CODE
```

BUFFER (79)

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

```
: BUFFER USE @ DUP >R BEGIN +BUF UNTIL USE !
         R@ @ 0< IF R@ 2+ R@ @ 7FFF AND 0 R/W THEN
         R@ ! R@ PREV ! R> 2+ ;
```

C! (79)

byte addr C! -> ----

Stores the low 8 bits of top of stack at address.

```
CODE C! assembly mnemonics END-CODE
```

C, (F)

byte C, -> ----

Stores the low 8 bits of top of stack into the next available dictionary byte, advancing the dictionary pointer.

```
: C, HERE C! 1 ALLOT ;
```

C/L (F)

A constant containing the number of characters per line of

FORTH text. This is set to 64 by convention. The terminal width is contained in the variable TW.

C@ (79)

addr C@ -> b

Leaves the 8 bit contents of memory address.

CODE C@ assembly mnemonics END-CODE

CCOLD (F)

The COLD start routine executed when system is booted from disk. This word does not reset the vocabulary link pointers and is not a suitable call after you have been working with the system.

CODE CCOLD assembler mnemonics END-CODE

CFA (F)

pfa CFA -> cfa

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

: CFA 2- ;

CLIT (F)

---- CLIT -> n

Within a colon-definition, CLIT is automatically compiled before each number between 0 and 255 encountered in the input text. Later execution of CLIT causes the contents of the next dictionary address to be pushed to the data stack.

CODE CLIT assembly mnemonics END-CODE

CLITERAL (I,M,C2)

byte CLITERAL -> ----

If compiling, then compile the low 8 bits of the stack as an 8 bit literal. This definition is IMMEDIATE so that it will execute during a colon definition. The intended use is:

: <name> ...[calculate] CLITERAL .... ;

: CLITERAL STATE @ IF COMPILE CLIT C, THEN ; IMMEDIATE

CLOAD (S)

n CLOAD -> ---

This command will force load the next consecutive screen on the input cassette to screen n, LIST screen n, and then LOAD screen n. The system variable DISK must be set to 0 before using this command.

: CLOAD DUP SGET BASE @ OVER LIST BASE ! LOAD ;

CMOVE (79)

from to count CMOVE -> ----

Moves the specified quantity of bytes beginning at address from to address to. The content of address from is moved first proceeding toward high memory.

CODE CMOVE assembly mnemonics END-CODE

CODE (F)

Used in the form:

CODE <name> .... END-CODE

Constructs a new dictionary entry <name>, sets CONTEXT to ASSEMBLER and assembles into machine code the assembler mnemonics .... which follow up to END-CODE . See END-CODE in the ASSEMBLER vocabulary.

: CODE ?EXEC !CSP CREATE LATEST PFA DUP
CFA ! SMUDGE AINIT ; IMMEDIATE

COLD (F)

The cold start procedure which truncates the dictionary at FENCE after first adjusting the vocabulary links. System parameters are reset according to the values in the bootup area following ORIGIN. The system is restarted via ABORT. May be called from the terminal to remove application programs and restart the system.

: COLD RVL CCOLD ;

COMPILE (79,C2)

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

: COMPILE ?COMP R> DUP 2+ >R @ , ;

CONSTANT (79)

n CONSTANT <name> -> ----

A defining word used to create word <name>, with its parameter field containing n. When <name> is later executed, it will push the value of n to the parameter stack.

```
: CONSTANT CREATE , ;CODE assembly mnemonics END-CODE
```

CONTEXT (U,79)

---- CONTEXT -> addr

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

CONVERT (79)

d1 addr1 CONVERT -> d2 addr2

Convert to the equivalent stack number the text beginning at addr+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 convertable character.

```
: CONVERT (NUMBER) ;
```

COUNT (79)

addr1 COUNT -> addr2 n

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

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

CR (79)

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

```
CODE CR assembly mnemonics END-CODE
```

CREATE (79)

---- CREATE <name> -> ----

A defining word used as shown above to create a dictionary entry for <name>, without allocating any parameter field

memory. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the data stack.

```
: CREATE FIRST HERE A0 + U< 2 ?ERROR -FIND
    IF DROP NFA ID. 4 MESSAGE SPACE THEN
    HERE DUP C@ WIDTH @ MIN 1+ ALLOT DUP
    80 TOGGLE HERE 1 - 80 TOGGLE LATEST ,
    CURRENT @ ! 2 ALLOT ;CODE mnemonics END-CODE
```

CSP (U,F)

---- CSP -> addr

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

CURRENT (U,79)

---- CURRENT -> addr

Leave the address of a variable specifying the vocabulary into which new word definitions are to be entered.

D+ (79)

d1 d2 D+ -> dsum

Leaves the double precision sum of the two double precision numbers d1 and d2.

```
CODE D+ assembly mnemonics END-CODE
```

D+- (F)

d1 n D+- -> d2

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

```
: D+- 0< IF DNEGATE THEN ;
```

D- (79)

d1 d2 D- -> d3

Subtract d2 from d1 and leave the difference as d3.

```
: D- DNEGATE D+ ;
```

D. (79)

d D. -> ----

Prints a signed double number from a 32 bit two's complement value. The high order 16 bits are most accessible on the

stack. Conversion is performed according to current BASE. A blank follows. Pronounced "dee-dot".

```
: D.  0 D.R SPACE ;
```

D.R (79)

```
d  n  D.R  ->  ----
```

Display d converted according to BASE, right aligned in an n character field. Display the sign only if negative.

```
: D.R  >R SWAP OVER DABS <# #S SIGN #> R>
       OVER - SPACES TYPE ;
```

D0< (S)

```
d  D0<  ->  flag
```

Leave a true flag if double number d is less than 0.

CODE D0< assembly mnemonics END-CODE

D0= (79)

```
d  D0=  ->  flag
```

Leave a true flag if the double number d is equal to zero.

CODE D0= assembly mnemonics END-CODE

D< (79)

```
d1  d2  D<  ->  flag
```

Leave a true flag if double number d1 is less than d2.

CODE D< assembly mnemonics END-CODE

D= (79)

```
d1  d2  D=  ->  flag
```

Leave a true flag if double number d1 is equal to d2.

```
: D=  D- D0= ;
```

D> (S)

```
d1  d2  D>  ->  flag
```

Leave a true flag if double number d1 is greater then d2.

```
: D>  2SWAP D< ;
```


DABS (79)

```
d  DABS  ->  ud
```

Leaves the absolute value ud of a signed double number.

```
: DABS  DUP D+- ;
```

DECIMAL (79)

Sets the numeric conversion BASE for decimal input-output.

```
: DECIMAL A BASE ! ;
```

DEFINITIONS (79)

Used in the form:

```
<vname>  DEFINITIONS
```

Sets the CURRENT vocabulary to the CONTEXT vocabulary. In the example executing vocabulary name <vname> made it the CONTEXT vocabulary and executing DEFINITIONS made both specify vocabulary <vname>.

```
: DEFINITIONS  CONTEXT @ CURRENT ! ;
```

DEPTH (79)

```
----  DEPTH  ->  n
```

Leave the number of the quantity of 16 bit values contained in the data stack, before n was added.

```
: DEPTH  SP@ 8E SWAP - 2 / ;
```

DIGIT (F)

```
char  n1  DIGIT  ->  n2  tf (valid conversion)
char  n1  DIGIT  ->  ff     (invalid conversion)
```

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

CODE DIGIT assembly mnemonics END-CODE

DISK (S)

A system variable that selects the disk system for mass storage when set to 1 and simulated disk in ram when set to 0. If you have a cassette only system setting DISK to 1 will result in a system crash!! If you have a disk system you must set DISK to 0 to use the cassette GET and PUT commands.

DLITERAL (I,F)

```
d   DLITERAL  ->  d     (executing)
d   DLITERAL  ->  ----  (compiling)
```

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

```
: DLITERAL  STATE @ IF SWAP [COMPILE] LITERAL
            [COMPILE] LITERAL THEN ; IMMEDIATE
```

DMAX (79)

```
d1  d2  DMAX  ->  d3
```

Leave the larger of the two double numbers d1 and d2 as d3.

```
: DMAX  2OVER 2OVER D< IF 2SWAP THEN 2DROP ;
```

DMIN (79)

```
d1  d2  DMIN  ->  d3
```

Leave the smaller of the two double numbers d1 and d2 as d3.

```
: DMIN  2OVER 2OVER D> IF 2SWAP THEN 2DROP ;
```

DNEGATE (79)

```
d   DNEGATE  ->  -d
```

Leave the double number two's complement of the double number d, ie., the difference 0 - d.

```
CODE  DNEGATE  assembly mnemonics END-CODE
```


DO (I,C2,79)

```
n1   n2   DO  ->  ----  (execute)

     DO   ->   addr   n   (compiling)
```

Occurs in a colon definition in the form:

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

At run-time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with an initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. Until the new index equals or exceeds the limit, execution loops back to just after DO, otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the result of other operations. Within a loop I will copy the current value of the index to the parameter stack, J the first outer loop and K the second outer loop. See I, LOOP, +LOOP, and LEAVE. When compiling within the colon definition, DO compiles (DO), leaves the following address addr and n for later error checking.

```
: DO   COMPILE (DO) HERE 3 ; IMMEDIATE
```

DOES> (I,C2+,79)

Define the run time action of a word created by a high level defining word. Used in the form:

```
: <name> ... CREATE ... DOES> ... ;
```

and then

```
<name>  <namex>
```

Marks the termination of the defining part of the defining word <name> and begins the definition of the run time action for words that will later be defined by <name>. On execution of <namex> the sequence of words between DOES> and ; will be executed, with the address of <namex>'s parameter field on the stack. Typical uses include the FORTH assembler, multi-dimensional arrays, and compiler generation.

```
: DOES>  COMPILE (;CODE) 20 C, [ HERE 0A + ]
         LITERAL , ; ASSEMBLER assembly mnemonics
```

DP (U,F)

```
----   DP  ->  addr
```

A user variable, the dictionary pointer, which contains the

address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.

DPL (U,F)

---- DPL -> addr

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

DROP (79)

n DROP -> ----

Drops the number n from top of parameter stack.

```
CODE  DROP assembly mnemonics END-CODE
```

DRW (S)

addr blk flag DRW -> ----

This is the disk mass storage read/write linkage called by by the word R/W when the variable DISK is set to 1. This word in not intended for user execution.

```
: DRW  >R S/BLK * DUP 8 < IF R> 7FFF PREV @ !
       6 ERROR THEN 8 - DUP 230 <
       IF R> SET2 (SET) EXT1 THEN
       230 2 / - DUP 230 < IF R> SET1 (SET)
       EXIT THEN R> 7FFF PREV @ ! 6 ERROR ;
```

DU< (79)

ud1 ud2 DU< -> flag

Leave a true flag if unsigned double number ud1 is less than unsigned double number ud2.

```
CODE  DU< assembly mnemonics END-CODE
```

DUP (79)

n DUP -> n n

Duplicates number on top of parameter stack.

```
CODE  DUP  assembly mnemonics END-CODE
```

EDITOR (I,F)

The name of the FORTH text editor vocabulary. Execution makes

EDITOR the CONTEXT vocabulary. The EDITOR assists in the development of disk screens.

ELSE (I,C2,79)

addr1 n1 ELSE -> addr2 n2 (compiling)

Occurs within a colon definition in the form:

IF .... ELSE .... THEN

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

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

```
: ELSE  2 ?PAIRS COMPILE BRANCH HERE 0 ,
        SWAP 2 [COMPILE] THEN 2 ; IMMEDIATE
```

EMIT (79)

char EMIT -> ----

Transmits ASCII character char to the selected output device. The variable OUT is incremented for each character output.

```
CODE  EMIT  assembly mnemonics END-CODE
```

EMPTY-BUFFERS (79)

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

```
: EMPTY-BUFFERS  FIRST LIMIT OVER - ERASE ;
```

ENCLOSE (F)

addr1 char ENCLOSE -> addr1 n1 n2 n3

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

```
CODE  ENCLOSE  assembly mnemonics END-CODE
```

ERASE (F)

addr n ERASE -> ----

Clears a region of memory to zero from addr for n bytes.

```
: ERASE 0 FILL ;
```

ERROR (F)

line ERROR -> in blk

Executes error notification and restarts system. WARNING is first examined. If 1, the text of line n, relative to screen 1 of the file assigned to the channel number stored in VCHAN is printed to the console device. If WARNING = 0, n is just printed as a message number. If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). SK-FORTH79 saves the contents of IN and BLK to assist in determining the location of the error. The data stack is reset and the values in and blk are located just below the first available entry. They can be extracted by using OVER OVER. See the definition WHERE for an example. The final action is the execution of QUIT.

```
: ERROR   WARNING @ 0< IF (ABORT) THEN HERE
          COUNT TYPE ." ? " MESSAGE SP! DROP DROP
          >IN @ BLK @ QUIT ;
```

EXECUTE (79)

addr EXECUTE -> ----

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

CODE EXECUTE assembly mnemonics END-CODE

EXIT (79)

When compiled within a colon definition, terminate execution of that definition at the point where EXIT occurs. May not be used within a DO....LOOP .

```
: EXIT R> DROP ;
```

EXPECT (79)

addr count EXPECT -> ----

Transfers characters from the terminal to successive addresses starting at addr, until a return is received from the terminal. One null is added at the end of the text. In SK-FORTH79 EXPECT is coded in machine language and contains the input line editor. By positioning the cursor at the desired point on your terminal screen and specifying count as the field width, you can program windowed input and protected fields. See also the section of the manual entitled Line Editing Commands.

CODE EXPECT assembly mnemonics END-CODE

FENCE (U,F)

---- FENCE -> addr

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

FILL (79)

addr n byte FILL -> ----

Fill memory beginning at address with a sequence of n copies of byte. If the quantity n is less than or equal to zero, take no action.

```
: FILL  OVER 0> IF SWAP >R OVER C! DUP 1+ R> 1- CMOVE
        ELSE DROP DROP DROP THEN ;
```

FIND (79)

---- FIND <name> -> addr

Leave the compilation address of the next word name, which is accepted from the input stream. If that word cannot be found in the dictionary after a search of CONTEXT and FORTH leave a 0 for addr.

```
: FIND -FIND IF DROP CFA ELSE 0 THEN ;
```

FIRST (F)

---- FIRST -> n

A constant that leaves the address of the first (lowest) block buffer. If you are going to reconfigure your system to allow more memory for user programs the value of FIRST must be altered. See section of the manual entitled Reconfiguration.

FLD (U,F)

---- FLD -> addr

A user variable for control of number output field width.

Presently unused in SK-FORTH79.

FORGET (79)

Executed in the form: FORGET <name>
Deletes definition named <name> from the dictionary and all entries physically following it. An error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same.

```
: FORGET [COMPILE] ' NFA DUP FENCE @ U< 15 ?ERROR
         >R VOC-LINK @ BEGIN R@ OVER U< WHILE
         [COMPILE] FORTH DEFINITIONS @ DUP VOC-LINK !
         REPEAT BEGIN DUP 4 - BEGIN PFA LFA @ DUP R@
         U< UNTIL OVER 2 - ! @ ?DUP 0= UNTIL R> DP ! ;
```

FORTH (I,79)

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. FORTH is an IMMEDIATE definition so it will execute even during the creation of a colon definition to select this vocabulary at compile time.

HARD (S)

```
----   HARD  ->  addr
```

A variable whose value indicates whether or not output is to be routed to the printer. If HARD has a value of 0 output is to the terminal only. If HARD has the value 1 output is to both the terminal and the hard copy device. If HARD has the value -1 output is to the hard copy device only.

HERE (79)

```
----   HERE  ->  addr
```

Leaves the address of the next available dictionary location.

```
: HERE  DP @ ;
```

HEX (F)

Sets the numeric conversion base to sixteen (hexadecimal).

```
: HEX  10 BASE ! ;
```

HLD (U,F)

```
----   HLD  ->  addr
```

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

HI (S)

A system variable containing the high memory bound of the simulated disk in RAM. See the section of the manual titled Reconfiguration for information on changing the value of this variable.

HOLD (79)

Used between <# and #> to insert an ASCII character into a pictured numeric output string. For example, 2E HOLD will place a decimal point.

```
: HOLD  -1 HLD +! HLD @ C! ;
```

I (79)

```
----   I  ->  n
```

Return the index of the inner most loop. May be used only within a DO...LOOP as follows to copy the loop index to the data stack.

```
DO....I....LOOP   or DO....I....n  +LOOP

CODE  I   assembly mnemonics END-CODE
```

ID. (F)

```
addr   ID.  ->  ----
```

Prints a definition's name from its name field address (NFA).

```
: ID.  PAD 20 5F FILL DUP PFA LFA OVER - PAD
       SWAP CMOVE PAD COUNT 1F AND TYPE SPACE ;
```

IF (I,C2,79)

```
flag   IF  ->  ----        (run-time)
----   IF  ->  addr  n     (compile)
```

Occurs in a colon definition in the form:

```
IF  (true part)  THEN
IF  (true part)  ELSE  (false part)  THEN
```

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

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

the offset and error testing.

```
: IF   COMPILE OBRANCH HERE 0 , 2 ; IMMEDIATE
```

IMMEDIATE (79)

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

```
: IMMEDIATE  LATEST 40 TOGGLE ;
```

INDEX (F)

```
from    to    INDEX  ->  ----
```

Prints the first line of each screen over the range from, to. This is used to view the comments which are usually on the first line of each screen describing their contents.

```
: INDEX CR 1+ SWAP DO CR I 3 .R SPACE 0 I .LINE
        ?TERMINAL IF LEAVE THEN LOOP ;
```

INTERPRET (F)

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

```
: INTERPRET BEGIN IF STATE @ < IF CFA ,
            ELSE CFA EXECUTE THEN ?STACK
            ELSE HERE NUMBER DPL @ 1+ IF [COMPILE]
            DLITERAL ELSE DROP DUP FF00 AND
            IF [COMPILE] LITERAL ELSE [COMPILE] CLITERAL
            THEN THEN ?STACK THEN AGAIN ;
```

J (79)

```
----   J  ->  n
```

Return the index of the first outer loop. May be used only within nested DO...LOOP's having the following form:

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



```
CODE  J  assembly mnemonics END-CODE
```

K (S)

Return the index of the second outer loop. May be used only within nested DO....LOOP's having the following form:

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

```
CODE  K  assembly mnemonics END-CODE
```

KEY (79)

```
----   KEY  ->  char
```

Leaves the ASCII value of the next terminal key struck. The character obtained is not echoed back to the terminal display.

```
CODE  KEY assembly mnemonics END-CODE
```

L2 (S)

```
n   L2  ->  flag
```

Low level high speed SYM cassette load routine. n is the file identification of the requested tape file. The flag is returned true for a bad load and false for an OK load. Tape motors are not operated by this command. L2 is used by the GET command. User would begin with L2 to custom design his own cassette interface. See also TAPE and R/T .

```
CODE  L2  assembly mnemonics  END-CODE
```

LATEST (F)

```
----   LATEST  ->  addr
```

Leaves the name field address (NFA) of the topmost word in the CURRENT vocabulary.

```
: LATEST  CURRENT @ @ ;
```

LEAVE (C,79)

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

```
CODE  LEAVE  assembly mnemonics END-CODE
```

LF2 (S)

```
n   addr1   addr2   LF2  ->  flag
```

Low level SYM high speed forced load to addr1, addr2. The tape identification n , must be $FF or an error results. Routine returns flag true for a bad load and false for a good load. This routine is used by the SGET command. Tape motors are not operated by this command. See also SGET , TAPE and R/T .

```
CODE   LF2  assembly mnemonics  END-CODE
```

LFA (F)

```
pfa    LFA  ->  lfa
```

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

```
: LFA  4 - ;
```

LIMIT (F)

```
----    LIMIT  ->  n
```

A system constant leaving the address just above the highest memory available for a disk buffer. The value of this constant must be changed if you plan to reconfigure the system to allot more space for user application programs. See the section of the manual titled Reconfiguration.

LIST (79)

```
n    LIST  ->  ----
```

Displays the ASCII text of screen n on the selected output device. SCR contains the screen number after the list is complete. If the variable HARD is set to 1 output will be also routed to the printer.

```
: LIST  DECIMAL CR DUP SCR ! ." Screen " DUP .  HEX
        5 .R ." hex" DECIMAL 10 0 DO CR I 3 .R SPACE
        I SCR @ .LINE ?TERMINAL IF LEAVE THEN LOOP CR ;
```

LIT (F)

```
----    LIT  ->  n
```

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

```
CODE  LIT  assembly mnemonics  END-CODE
```


LITERAL (I,C2,79)

```
n    LITERAL  ->  ----    (compiling)
```

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

```
: <name>  [  calculate  ]  LITERAL  ;
```

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

```
: LITERAL  STATE @ IF COMPILE LIT , THEN ; IMMEDIATE
```

LO (S)

A system variable containing the low memory bound of the simulated disk in ram screens. See the section of the manual titled Reconfiguration for information on changing this value.

LOAD (79)

```
scr    LOAD  ->  ----
```

Begins interpretation of screen scr. Loading will terminate at the end of the screen, at a ;S or at an error. If an error occurs use WHERE to be notified of the screen location of the error. See also ;S , --> .

```
: LOAD  BLK @ >R >IN @ >R 0 >IN ! B/SCR *
        BLK ! INTERPRET R> >IN ! R> BLK ! ;
```

LOOP (I,C2,79)

```
----      LOOP  ->  ----    (executing)
addr  n   LOOP  ->  ----    (compiling)
```

Occurs in a colon definition in the form:

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

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and loop 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 which time the parameters are discarded and execution continues ahead.

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

```
: LOOP  3 ?PAIRS COMPILE (LOOP) BACK ; IMMEDIATE
```

M* (F)

n1 n2 M* -> d

A mixed magnitude math operator which leaves the double integer signed product of two signed numbers.

```
: M*  OVER OVER XOR >R ABS SWAP ABS U* R> D+- ;
```

M/ (F)

d n1 M/ -> n2 n3

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

```
: M/  OVER >R >R DABS R@ ABS U/MOD R> R@
      XOR +- SWAP R> +- SWAP ;
```

M/MOD (F)

ud1 u2 M/MOD -> u3 ud4

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

```
: M/MOD  >R 0 R@ U/MOD R> SWAP >R U/MOD R> ;
```

MAX (79)

n1 n2 MAX -> max

Leaves the greater of the two numbers n1 and n2.

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

MDISK (S)

---- MDISK -> ----

All communication with the disk operating system is through the word MDISK which in turn calls (MDISK). The difference between these words is that MDISK also checks for disk access errors. Both MDISK and (MDISK) assume that the command and parameters are first set in FODS parameter area ($7200-7220 for SYM-FODS and $F200-F220 for KIM-FODS). The word (SET) does this for the standard FORTH disk linkage which uses R/W and DRW .

```
: MDISK  (MDISK) 721B C@ FF < IF HEX 721B C@
         .  OFF 8 ERROR THEN ;
```


MESSAGE (F)

n MESSAGE -> ----

Prints on the selected output device the text of line n relative to screen 1 of the virtual memory file assigned to channel number stored in VCHAN. The value of n must be >= 0. MESSAGE may be used to print incidental text such as report headers. WARNING must be set to 1 for the messages to be printed. If WARNING is 0 only the message numbers will be printed.

```
: MESSAGE  WARNING @ IF ?DUP IF .LINE THEN
           ELSE ." MSG # " .  THEN ;
```

MIN (79)

n1 n2 MIN -> min

Leaves the smaller of the numbers n1 and n2.

```
: MIN  OVER OVER > IF SWAP THEN DROP ;
```

MOD (79)

n1 n2 MOD -> mod

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

```
: MOD  /MOD DROP ;
```

MON (S)

Exit to the KIM or SYM monitor. For the SYM, return to FORTH via .G (ret) ; for KIM push the space bar and the G key. MON can be used within a colon definition to exit a running program. If operations while in the monitor do not disturb the stack or any of FORTH's memory space the program will resume execution just ahead of MON. Note that the faster control S of the line editor returns via the warm start entry point so that the state of the system may be different from when the exit occurred.

MOVE (79)

addr1 addr2 n MOVE -> ----

Move n 16 bit memory words from memory beginning at addr1 to memory at addr2. The word stored at addr1 is moved first. If n is negative or zero, nothing is moved.

```
: MOVE  2* CMOVE ;
```

MVAR (S)

---- MVAR -> addr

A variable used by the assembler which is initialized by CODE and ;CODE to hold the offset ($0D) to be added to the base opcode to produce absolute addressing. This variable is modified by the assembler addressing modes and reset to $0D by the word SMVAR after the assembly of each instruction.

MYSELF (I,C,S)

Used within a colon definition to compile the code field address (CFA) of the word curently being defined. MYSELF is used to construct recursive definitions. Use would take the following form:

```
: <name> .... MYSELF .... ;
```

In the example above MYSELF will compile the code field address of <name>.

```
: MYSELF LATEST PFA CFA , ; IMMEDIATE
```

NBUF (S)

---- NBUF -> n

A system constant which leaves the number of disk buffers as n.

NEGATE (79)

n NEGATE -> -n

Leave the two's complement of a number, ie., the difference of 0 less n.

```
CODE NEGATE assembly mnemonics END-CODE
```

NFA (F)

pfa NFA -> nfa

Converts the parameter field address of a definition to its name field address. See also CFA, LFA, and PFA.

```
: NFA 5 - -1 TRAVERSE ;
```

NOT (79)

flag1 NOT -> flag2

Reverse the boolean value of flag1 and leave as flag2. This is identical to 0= .

```
: NOT 0= ;
```

NUMBER (F)

addr NUMBER -> d

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

```
: NUMBER 0 0 ROT DUP 1+ C@ 2D = DUP >R + -1
         BEGIN DPL ! CONVERT DUP C@ BL -
         WHILE DUP C@ 2E - 0 ?ERROR 0 REPEAT
         DROP R> IF DNEGATE THEN ;
```

OFF (S)

This word turns the disk drives off if a disk system is present. If user has no disk system it does nothing.

```
: OFF DISK @ IF 20 7218 C! (MDISK) THEN ;
```

ON (S)

This word turns the disk drives on if a disk system is present. If no disk system is present it does nothing.

```
: ON DISK @ IF 30 7218 C! (MDISK) THEN ;
```

OR (79)

n1 n2 OR -> or

Leaves the bit-wise logical OR of the single numbers n1 and n2.

```
CODE OR assembly mnemonics END-CODE
```

OUT (U,F)

---- OUT -> addr

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

OVER (79)

n1 n2 OVER -> n1 n2 n1

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

```
CODE  OVER  assembly mnemonics END-CODE
```

PAD (79)

```
----    PAD   ->   addr
```

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

```
: PAD  HERE 54 + ;
```

PFA (F)

```
nfa    PFA   ->   pfa
```

Converts the name field address of a compiled definition to its parameter field address. See also NFA, LFA, and CFA.

```
: PFA  1 TRAVERSE 5 + ;
```

PICK (79)

```
n1    PICK  ->   n2
```

Return the contents of the n1-th stack value, not countine n1 itself to the top of the data stack as n2. An error condition results for n less than 1 or for a indexing past the bottom of the stack.

```
2 PICK = OVER        1 PICK = DUP

: PICK   ?STK IF DROP 1 ERROR ELSE 2* SP@ + @ ;
```

PREV (F)

```
----    PREV   ->   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.

PUT (S)

```
n1    n2    PUT   ->   ----
```

High level high speed cassette output routine. Screen n2 is saved to the output cassette with file identification given by n1. The output cassette is specified by the variable W/T. Tape motor is automatically turned on and off. It is assumed that the variable DISK is set to 0 indicating simulated disk in ram screens.

```
: PUT  SAVE-BUFFERS SCRAD TEAD ! TSAD ! TFID !
       1 W/T @ TAPE S2 0 W/T @ TAPE BEEP A ?ERROR ;
```


QUERY (79)

Inputs 80 characters of text (or until a "return") from the operator's terminal. Text is positioned at the address contained in TIB with >IN set to zero. See also EXPECT.

```
: QUERY  TIB @ TW EXPECT 0 >IN ! ;
```

QUIT (79)

Clears the return stack, stops compilation, and returns control to the operator's terminal. No message is given.

```
: QUIT  0 BLK ! [COMPILE] [ BEGIN RP! CR QUERY
        INTERPRET STATE @ 0= IF ." OK" THEN AGAIN ;
```

R# (U,F)

```
----    R#   ->   addr
```

A user variable which may contain the location of an editing cursor, or other file related function. Currently not used by SK-FORTH79 but could be utilized to keep track of a second cursor for visible editing.

R/T (S)

A system variable whose value indicates which cassette is to be used for input by the high level cassette commands GET and SGET. A value of zero indicates the standard SYM-1 remote controlled cassette. A value of 1 indicates the second cassette connected as described in the RAE-1 Manual.

R/W (F)

```
addr    blk    flag    R/W   ->   ----
```

The standard FORTH disk read/write linkage. addr specifies the source or destination block buffer, and blk is the sequential number of the referenced block. If the flag has a value of 0 then a write operation is indicated, if the flag has the value 1 then a read operation is indicated. R/W determines the location on mass storage, performs the read/write, and performs any error checking.

```
: R/W  DISK @ IF DRW ELSE TRW THEN ;
```

R> (79)

```
----    R>   ->   n
```

Removes the top value from the return stack and leaves it on the parameter stack. See also R@ and >R .

CODE R> assembly mnemonics END-CODE

R@ (79)

---- R@ -> n

Copy the number on the top of the return stack to the data stack.

CODE R@ assembly mnemonics END-CODE

RDEPTH (S)

---- RDEPTH -> n

Determine the depth of the return stack and leave as n on the data stack.

CODE RDEPTH assembly mnemonics END-CODE

REPEAT (I,C2,79)

addr n REPEAT -> ---- (compiling)

Used within a colon definition in the form:

.... BEGIN .... WHILE .... REPEAT ....

At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN. At compile time, REPEAT compiles BRANCH and the offset from HERE to addr. n is used for error testing.

```
: REPEAT >R >R [COMPILE] AGAIN R> R> 2-
        [COMPILE] THEN ; IMMEDIATE
```

ROLL (79)

n ROLL -> ---- (stack rolled)

Extract the n-th data stack value to the top of the stack, not counting n itself, moving the remaining values into the vacated position. An error condition results if n exceeds the stack depth or is less than one.

```
3 ROLL = ROT

2 ROLL = SWAP

: ROLL ?STK IF DROP 1 ERROR ELSE (ROLL) THEN ;
```

ROT (79)

n1 n2 n3 ROT -> n2 n3 n1

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

```
: ROT >R SWAP R> SWAP ;
```

RP! (F)

Initializes the return stack pointer from the silent user variable R0.

CODE RP! assembly mnemonics END-CODE

RVL (E,S)

Restore vocabulary links so that dictionary can be truncated at FENCE. This word shares part of the code in FORGET and is not intended for user execution. It is used in COLD so that the FORTH system can be correctly cold started after new vocabularies have been defined.

```
: RVL FENCE @ [ ' FORGET 4 + ] LITERAL >R ;
```

S->D (F)

n S->D -> d

Sign extends a single number to form a double number.

```
: S->D DUP 0< NEGATE ;
```

S. (S)

Nondestructive stack print. Prints the stack as signed 16 bit numbers identifying the top and botom of the stack.

```
: S. ." BOT> " DEPTH ?DUP IF 1 SWAP DO I PICK .
                -1 +LOOP THEN ." <TOP " ;
```

S/BLK (S)

A system constant containing the number of sectors per block buffer. The default value of this constant is 8.

S2 (S)

n1 addr1 addr2 S2 -> flag

Low level high speed cassette save routine. The file is put to tape with identification n1 , addr1 and addr2 are start and end addresses respectively. Routine returns flag true for a bad save and false for a good save. Tape motors are not operated by this command. S2 is used by PUT and SYSSAVE .

CODE S2 assembly mnemonics END-CODE

SAVE-BUFFERS (79)

Write all blocks to mass storage that have been flaged as UPDATEd. An error condition results if mass storage writing is not completed.

```
: SAVE-BUFFERS NBUF 1+ 0 DO 7FFF BUFFER DROP LOOP ;
```

SCR (U,79)

```
---- SCR -> addr
```

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

SCRAD (S)

```
n SCRAD -> addr1 addr2
```

Produces the start and end addresses of the simulated disk screen n as addr1 and addr2 respectively. This word uses the system variables HI and LO for generation and checking the addresses of the requested screen.

```
: SCRAD 1- 400 * LO @ + HI @ OVER U< 6 ?ERROR
        DUP 3FF + BEEP ;
```

SET1 (S)

```
---- SET1 -> ----
```

This word selects drive 1 as the default drive for disk transactions.

```
: SET1 DISK @ IF 0 7215 C! 0 7214 C! THEN ;
```

SET2 (S)

This word selects drive 2 as the default drive for disk transactions.

```
: SET2 DISK @ IF 2 7215 C! 0 7214 C! THEN ;
```

SGET (S)

```
n SGET -> ----
```

Force loads the next tape file from the current input cassette disregarding the file identification and address information on the tape. The file is loaded into the disk simulated ram screen n. An error will result if the file is not exactly 1K bytes long. This word uses LF2.


```
: SGET SAVE-BUFFERS FF TFID ! SCRAD TEAD ! TSAD !
       1 R/T @ TAPE LF2 0 R/T @ TAPE BEEP
       B ?ERROR EMPTY-BUFFERS ;
```

SIGN (79)

```
n d SIGN -> 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 #>.

```
: SIGN ROT 0< IF 2D HOLD THEN ;
```

SMUDGE (F)

Used during word definition to toggle the "SMUDGE bit" in a definition's name field. This prevents an uncompleted definition from being found during dictionary searches until compiling is complete without error. If the most recently created definition resulted in a compiling error and cannot be forgotten by FORGET, execute SMUDGE and try again.

```
: SMUDGE LATEST 20 TOGGLE ;
```

SMVAR (S)

```
---- SMVAR -> ----
```

Reset the assembler address mode offset variable MVAR to $0D, the value for absolute addressing.

```
: SMVAR 0D MVAR ! ;
```

SP! (F)

A procedure used to initialize the stack pointer from silent user variable S0.

```
CODE SP! assembly mnemonics END-CODE
```

SP@ (F)

```
---- SP@ -> addr
```

A procedure which returns the address of the parameter stack pointer to the top of the parameter stack as it was before SP@ was executed.

```
CODE  SP@  assembly mnemonics END-CODE
```

SPACE (79)

Transmits one ASCII blank to the output device.

```
: SPACE  BL EMIT ;
```

SPACES (79)

```
n   SPACES  ->  ----
```

Transmits n ASCII blanks to the output device.

```
: SPACES  0 MAX ?DUP IF 0 DO SPACE LOOP THEN ;
```

STATE (U,79)

```
----   STATE  ->  addr
```

A user variable containing the compilation state. A non zero value indicates compilation.

SWAP (79)

```
n1   n2   SWAP  ->  n2   n1
```

Exchanges the top two values on the parameter stack.

```
CODE  SWAP  assembly mnemonics END-CODE
```

SYSCOLD (S)

```
----   SYSCOLD  ->  ----
```

Resets the system and modifies the parameters in the boot up area to retain all words created since the last cold start. Usually followed by a SYSSAVE.

```
: SYSCOLD  [COMPILE] FORTH CONTEXT @ @ [ 2A +ORIGIN ] !
           HERE DUP [ 3A +ORIGIN ] ! [ 3C +ORIGIN ] !
           VOC-LINK @ [ 3E +ORIGIN ] ! CCOLD ;
```

SYSSAVE (S)

```
n   SYSSAVE  ->  ----
```

Dumps the entire FORTH system to the current output cassette with n for the file identification. If the system has been expanded and you want to save its current state than do a

SYSCOLD first, otherwise you will save the system as it was at the last cold start. If you only want to make a backup copy of the system then type COLD first. Users of disk systems should type SYSCOLD, get the last dictionary address with HEX HERE . and then exit to FODS to save the system.

```
: SYSSAVE  TFID ! 200 TSAD ! HERE TEAD ! 1 W/T @ TAPE
           BEEP S2 0 W/T @ TAPE BEEP A ?ERROR ;
```

TAPE (S)

```
n1   n2   TAPE  ->  ----
```

This word controls the cassete motors. n2 selects the cassette, 0 is the standard recorder and 1 is the optional recorder. If n1 is 1 the motor is turned on, if n1 is 0 the motor is turned off.

```
CODE  TAPE  assembly mnemonics  END-CODE
```

TEAD (S)

A system constant that points to the cassette file end address.

TFID (S)

A system constant that points to the cassette file identification.

TSAD (S)

A system constant that points to the cassette file start address.

THEN (I,C0,79)

```
addr   n   THEN  ->  ----  (compiling)
```

Occurs in a colon definition in the form:

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

At run time, THEN serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure.

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

```
: THEN ?COMP 2 ?PAIRS HERE OVER - SWAP ! ; IMMEDIATE
```

TIB (U,79)

---- TIB -> addr

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

TOGGLE (F)

addr b TOGGLE -> ----

Complements the contents of addr by the bit pattern b .

CODE TOGGLE assembly mnemonics END-CODE

TRACECOLON (S)

After this word executes all colon definitions will be traced at the terminal. Form of the trace is:

: <name> where <name> is the name of executing colon definition.

To stop tracing hit the break key, to exit trace mode hit RETURN, and to continue tracing type SPACE.

CODE TRACECOLON assembly mnemonics END-CODE

TRACEOFF (S)

When TRACEOFF is executed the trace feature will be turned off. Intended use is shown below:

.... TRACEON .... TRACEOFF ....

By using TRACEON and TRACEOFF you can trace only those parts of your program which are giving you trouble.

CODE TRACEOFF assembly mnemonics END-CODE

TRACEON (S)

When this word executes both colon and code definitions will be traced at the terminal. Format of the trace is

iiii nnnn wwww pp rrrr

where iiii is the high level interpretive pointer

nnnn is the word name.
wwww is the code field pointer to actual machine code
pp is pointer to top of parameter stack in page zero
rrrr is pointer to top of the return stack in page one

To stop tracing hit break key, to exit tracemode type RETURN and to continue the trace type SPACE.

CODE TRACEON assembly mnemonics END-CODE

TRAVERSE (F)

addr1 n TRAVERSE -> addr2

Moves across the name field of a MTU-FORTH79 variable length name field. addr1 is the address of either the length byte or the last letter. If n=1 the motion is toward hi memory, and if n=-1 the motion is toward low memory. The addr2 resulting is address of the other end of the name.

```
: TRAVERSE SWAP BEGIN OVER + 7F OVER C@ < UNTIL
           SWAP DROP ;
```

TRIAD (F)

scr TRIAD -> ----

Displays on the selected output device the three screens which begin with that numbered scr. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen 1. If HARD is set to 1 before TRIAD then output will be routed to the printer.

```
: TRIAD DUP 3 + SWAP DO CR I LIST LOOP
        CR 0F MESSAGE ;
```

TRW (S)

addr blk flag TRW -> ----

The standard disk linkage for the simulated disk in ram screens. TRW is called by R/W if the system variable DISK is set to 0.

```
: TRW  R>  DUP B/SCR < IF R> 7FFF PREV @ !
       6 ERROR THEN B/SCR - B/BUF * LO @ +
       HI @ B/BUF - OVER U< IF R> 7FFF PREV @ !
       6 ERROR THEN R> IF SWAP THEN B/BUF CMOVE ;
```

TYPE (79)

addr n TYPE -> ----

Transmits n characters from addr to the selected output device.

```
: TYPE ?DUP IF OVER + SWAP DO I C@ EMIT LOOP
               ELSE DROP THEN ;
```

TW (S)

---- TW -> n

A system constant that returns the output character width, n , of the console device.

U* (79)

u1 u2 U* -> ud

Leaves the unsigned double number product of the two unsigned numbers.

CODE U* assembly mnemonics END-CODE

U. (79)

un U. -> ----

Display un converted according to BASE as an unsigned number, in a free field format, with one trailing blank.

```
: U.  0 D. ;
```

U/MOD (79)

ud u1 U/MOD -> u2 u3

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

CODE U/MOD assembly mnemonics END-CODE

U< (79)

un1 un2 U< -> flag

Leave a true flag if the 16 bit unsigned integer un1 is less than the unsigned integer un2.

CODE U< assembly mnemonica END-CODE

UNTIL (I,C2,79)

```
flag    UNTIL  ->  ----   (run time)
addr n  UNTIL  ->  ----   (compiling)
```

Occurs within a colon definition in the form:

.... BEGIN .... UNTIL ....

At run time, UNTIL controls the conditional branch back to the correspondin BEGIN. If the flag is false, execution returns to just after BEGIN. If the flag is true, execution continues ahead. At compile-time, UNTIL compiles (OBRANCH) and an offset from HERE to addr. n is used for error tests.

```
: UNTIL  1 ?PAIRS COMPILE OBRANCH BACK ; IMMEDIATE
```

UPDATE (79)

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

```
: UPDATE  PREV @ @ 8000 OR PREV @ ! ;
```

USE (F)

---- USE -> addr

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

USER (F)

A defining word used in the form

n USER <uname>

which creates a user variable <uname>. The parameter field of <uname> contains n as a fixed offset relative to the user pointer register UP for this user variable. When <uname> 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.

```
CODE  USER  assembly mnemonics END-CODE
```

VARIABLE (79)

A defining word used in the form:

```
VARIABLE  <vname>
```

When VARIABLE is executed, it creates the definition <vname> with its parameter field initialized to 0. The application program must initialize the stored value. When <vname> is later executed, the address of its parameter field is left on the data stack, so that a fetch or store may access this location.

```
: VARIABLE  CREATE 0 , ;
```

VLIST (F)

Lists the names of the definitions in the context vocabulary. Hit break key to stop listing, followed by RETURN to quit or SPACE to continue listing.

```
: VLIST  80 OUT ! CONTEXT @ @ BEGIN OUT @ TW 10 - >
         IF CR 0 OUT ! THEN DUP ID.  SPACE SPACE
         PFA LFA @ DUP 0= ?TERMINAL OR UNTIL DROP ;
```

VOC-LNK (U,F)

```
----   VOC-LINK  ->  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 FORGETing through multiple vocabularies.

VOCABULARY (79)

A defining word used in the form

```
VOCABULARY  <vname>
```


to create a vocabulary definition <vname>. Subsequent use of <vname> will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence " <vname> DEFINITIONS " will also make <vname> the CURRENT vocabulary into which new definitions are placed. In MTU-FORTH79, <vname> will be so chained as to include all definitions of the vocabulary in which <vname> itself is defined. All vocabularies ultimately chain to FORTH. By convention vocabulary names are to be declared IMMEDIATE. See VOC-LINK.

```
: VOCABULARY  CREATE A081 , CURRENT @ CFA , HERE
              VOC-LINK @ , VOC-LINK ! DOES> 2+ CONTEXT ! ;
```

WARNING (U,F)

```
----   WARNING  ->  addr
```

A user variable containing a value to control messages. If WARNING is 1 then disk is selected and screen 1 is the base location for messages. If WARNING is 0 then no disk is present and messages will be presented by number only. If WARNING is -1 then (ABORT) is executed. (ABORT) can be modified for user specified error recovery. See also MESSAGE and ERROR.

WHERE (F)

If an error is encountered when executing LOAD then executing WHERE will print the offending screen number and the offending line marking the error.

```
: WHERE  OVER OVER B/SCR / DUP SCR ! ." Screen "
         DECIMAL .  CR C/L /MOD SWAP 2+ HERE C@ -
         SPACES ." @/" [COMPILE] EDITOR DUP CR
         3 .R SPACE SCR @ .LINE ;
```

WHILE (I,79)

```
   flag     WHILE  ->  ----                        (run time)
addr1 n1  WHILE  ->  addr1 n1 addr2 n2            (compiling)
```

Occurs in a colon definition in the form:

```
.... BEGIN .... WHILE .... REPEAT ....
```

At run time, WHILE selects conditional execution based on the flag. If the flag is true (non zero), WHILE continues execution of the true part through to REPEAT, which then branches back to BEGIN. If the flag 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.

```
: WHILE [COMPILE] IF 2+ ; IMMEDIATE
```

WIDTH (U,F)

```
---- WIDTH -> addr
```

A user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 through 31. The name character count and its natural characters are saved, up to the value of WIDTH. The value may be changed at any time within the above limits.

WORD (79)

```
char WORD <text> -> addr
```

Reads the next text characters from the input stream being interpreted until a delimiter char is found, storing the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte, then the characters, and ends with two or more blanks. Leading occurences of char are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disk block stored in BLK. The address of the beginning of this packed string is left on the data stack as addr. See also >IN and BLK.

```
: WORD BLK @ IF BLK @ BLOCK ELSE TIB @ THEN
       >IN @ + SWAP ENCLOSE HERE 22 BLANKS >IN +!
       OVER - >R R@ HERE C! + HERE 1+ R> CMOVE HERE ;
```

XOR (79)

```
n1 n2 XOR -> xor
```

Leaves the bitwise logical exclusive OR of the single numbers n1 and n2.

```
CODE XOR assembly mnemonics END-CODE
```

[(I,79)

Used in a colon definition in the form:

```
: <name> .... [ words ] .... ;
```

Suspend compilation. The words after [are executed, not compiled. This allows calculation or compilation exceptions

before resuming compilation with]. See LITERAL and] .

```
: [ 0 STATE ! ; IMMEDIATE
```

[COMPILE] (I,79)

Used in a colon definition in the form:

```
: <name> .... [COMPILE] FORTH .... ;
```

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

```
: [COMPILE] -FIND 0= 0 ?ERROR DROP CFA , ; IMMEDIATE
```

] (79)

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

```
: ] 0C0 STATE ! ;
```

FORTH BIBLIOGRAPHY

The primary source of information and applications with direct application to SK-FORTH79 is our own newsletter Saturn Softnews. Saturn Softnews is a quarterly publication of Saturn Software Limited.

Subscription Rates:

Volume 1 (Four issues beginning with June 1, 1981)
$10 US/$12 CAN (+$4 for overseas mailing).

Volume 2 (Four issues planned beginning with June 1, 1982)
$14 US/$16CAN (+$4 for overseas mailing).

Mail your subscriptions to:

SATURN SOFTWARE LIMITED, P. O. BOX 397,
NEW WESTMINSTER B. C. , V3L 4Y7, CANADA

Most of the manuals and books that follow are available either from:

FORTH INTEREST GROUP, P. O. BOX 1105, SAN CARLOS, CA 94070
OR
MOUNTAIN VIEW PRESS, P. O. BOX 4656, MOUNTAIN VIEW, CA 94040

We suggest you write to both groups for their product list and order form with their current prices.

1. 6502 FIG FORTH Installation Manual- complete language model, glossary, and installation instructions.

2. 6502 Assembly source listing of fig-FORTH- requires customization to particular system. Assembler and cassette interface not included.

3. FORTH 79-STANDARD- Publication of the FORTH standards team.

4. Using FORTH, by FORTH, Inc. - This is claimed to be the best users manual available (160 pages)

5. Kitt Peak Primer, by W. R. Stevens (275 pages, student exercises, some differences from fig-FORTH)

6. The CALTECH FORTH Manual, by M. S. Ewing (100+ pages, good on FORTH structure)

7. URTH Tutorial Manual, by Berg, Forsley, Hardwick (70 pages, URTH is the University of Rochester implementation of FORTH)

8. SYSTEMS GUIDE TO FIG-FORTH, by C. H. Ting (148 pages, details of the inner workings of fig-FORTH, 6 page errata included!)

9. Threaded Interpretive Languages, by R. G. Loeliger (250 pages, hard cover, an adaption of FORTH to the Z80)

10. Byte Magazine, Aug. 1980, The FORTH Issue

11. Byte Magazine, Sept. 1980, page 206, Varieties of Threaded Code for Language Implementation by T. Ritter and G. Walker

12. Byte Magazine, Oct. 1980, page 274, The FORTH Standards Team by W. R. Ragsdale.

13. Byte Magazine, Feb. 1981, page 152, Stacking Strings for FORTH by J. Cassady.

14. Byte Magazine, Mar. 1981, page 155, A Coding Sheet for FORTH

15. Dr. Dobb's Journal, Number 25, page 21 FORTH for Microcomputers by John James.

16. Dr. Dobb's Journal, Number 28, page 26, FORTH Dump Programs by John James.

17. Dr. Dobb's Journal, Number 46, page 25, Adding Arrays to FORTH by Ralph Deane.

18. Dr. Dobb's Journal, Number 50, page 40, A Proposal on Strings for FORTH, by Ralph Deane.

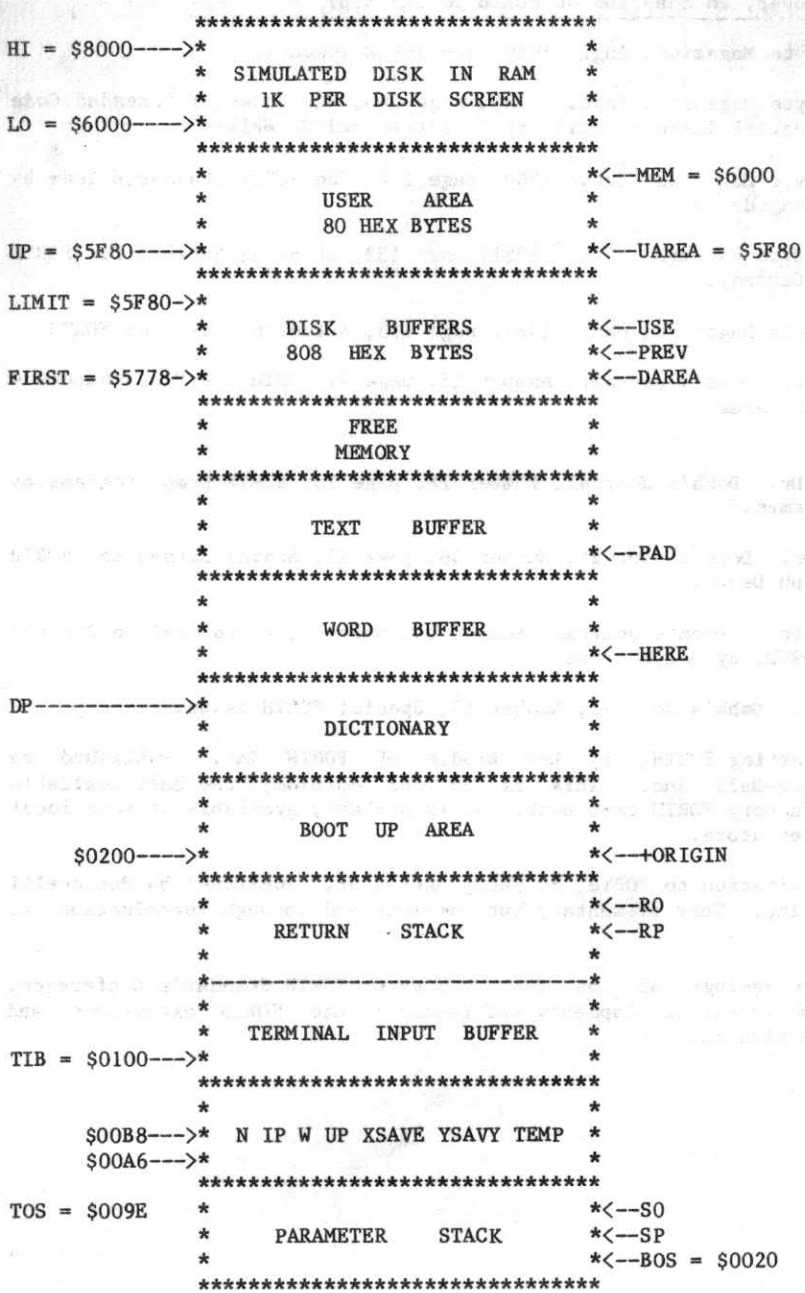
19. Dr. Dobb's Journal, Number 59, Special FORTH Issue.

20. Starting FORTH, by Leo Brodie of FORTH Inc. Published by Prentice-Hall Inc. This is in our opinion, the best available introductory FORTH text book. It is probably available at your local computer store.

21. Invitation to FORTH, by Harry Katzan Jr. Published by Petrocelli Books Inc. Very elementary but complete and through introduction to FORTH.

22. Proceedings of the 1981 Rochester FORTH Standards Conference. For the latest developments and research into FORTH extensions and standardization.

MEMORY MAP FOR 32K-CASSETTE SYSTEM




```
Screen 4      4 hex
  0 ( SK-FORTH 6502 ASSEMBLER-1                          J.W.B.  26:09:81 )
  1 FORTH DEFINITIONS
  2 : ATASK ;  ( DUMMY TO MARK START OF ASSEMBLER )
  3
  4 ASSEMBLER DEFINITIONS  HEX     ( START ASSEMBLER DEFINITIONS    )
  5
  6 ( INSTALLATION DEPENDENT CONSTANTS )
  7
  8   AE CONSTANT IP        B1 CONSTANT W        A6 CONSTANT N
  9   B5 CONSTANT XSAVE     B6 CONSTANT YSAVE    B3 CONSTANT UP
 10  25E CONSTANT PUT      263 CONSTANT NEXT    25C CONSTANT PUSH
 11  6A2 CONSTANT PUSHOA   433 CONSTANT POP     431 CONSTANT POPTWO
 12   33E CONSTANT SETUP
 13
 14
 15

Screen 5      5 hex
  0 ( SK-FORTH 6502 ASSEMBLER-2                          J.W.B  26:09:81 )
  1 ( nl  OPCHK  -->  flag            1 if valid opcode 0 otherwise )
  2 CREATE OPCHK 00B5 ,  B586 ,     ( CREATE AN ENTRY CALLED OPCHK  )
  3 4AB8 , 0990 , 4A  C, 44B0 , 22C9 , 40F0 , 3A50 , 904A , 4A12 ,
  4 0690 , 13C9 , 34F0 , 2E50 , 4A  C, 23B0 , 0AC9 , 27F0 , 2950 ,
  5 B04A , 4A09 , 1FB0 , 08C9 , 1FF0 , 1950 , 4A  C, 1AF0 , 08B0 ,
  6 0D29 , 04C9 , 06D0 , 1050 , 09C9 , 0CF0 , 904A , 4A05 , 02C9 ,
  7 04D0 , 00A9 , 02F0 , 01A9 , B5A6 , 4C  C, PUSHOA ,
  8  OPCHK DUP CFA !   ( PUT PFA IN CFA )
  9 : OPREL CREATE C, DOES> ( OPREL - TO CREATE RELATIVE OPCODES    )
 10        C@ C, DUP FF > IF HERE 1+ - DUP DUP 80 > SWAP -80 < OR
 11        IF 0 HERE !  1 1A ?ERROR THEN FF AND THEN C, ;
 12
 13 90 OPREL BCC,  B0 OPREL BCS,  F0 OPREL BEQ,  D0 OPREL BNE,
 14 30 OPREL BMI,  10 OPREL BPL,  50 OPREL BVC,  70 OPREL BVS,
 15

Screen 6      6 hex
  0 ( SK-FORTH  6502 ASSEMBLER-3                         J.W.B  26:09:81 )
  1 ( CREATE ADDRESS MODES )
  2
  3 : MODE CREATE C, DOES> C@ MVAR ! ;
  4
  5 09 MODE IM, 0D MODE AB, 05 MODE ZP, 1D MODE X,
  6 01 MODE IX, 11 MODE IY, 1D MODE AX, 19 MODE Y,
  7 19 MODE AY, 15 MODE ZX, 15 MODE ZY,
  8
  9 : EXCODE  MVAR  @  DUP  1D  =  OVER  0D  =  OR
 10         IF  3  PICK  FF00  AND  0=
 11         IF  DUP  8  -  MVAR  !  THEN  THEN
 12       DROP MVAR   @  SWAP  C@  OVER  +  OPCHK
 13         1B ?ERROR  C, DUP  0D  =  SWAP  18  >  OR
 14         IF  ,  ELSE  C,  THEN  SMVAR ;
 15
```

```
Screen 7    7 hex
 0 ( SK-FORTH  6502 ASSEMBLER-4                    J.W.B.  26:09:81 )
 1
 2 ( BUILD TWO AND THREE BYTE OPCODES                                )
 3
 4 : OPCODE CREATE C, DOES> EXCODE ;
 5 : OPXY  CREATE  ,  DOES>  MVAR  @  DUP
 6         9  =  IF  1  MVAR  !  THEN
 7        19  =  IF  1+  THEN  EXCODE  ;
 8 60 OPCODE ADC,  20 OPCODE AND,  01 OPCODE ASL,  1F OPCODE BIT,
 9 C0 OPCODE CMP,  C1 OPCODE DEC,  40 OPCODE EOR,  E1 OPCODE INC,
10 A0 OPCODE LDA,  41 OPCODE LSR,  00 OPCODE ORA,  21 OPCODE ROL,
11 61 OPCODE ROR,  E0 OPCODE SBC,  80 OPCODE STA,  81 OPCODE STX,
12 7F OPCODE STY,
13 E3DF OPXY CPX,        A5A1 OPXY LDX,
14 C3BF OPXY CPY,        A39F OPXY LDY,
15

Screen 8     8 hex
 0 ( SK-FORTH  6502 ASSEMBLER-5                    J.W.B.  26:09:81 )
 1 ( BUILD ONE BYTE OPCODES                                          )
 2 : OPBYT CREATE C, DOES> C@ C, ;
 3 0A OPBYT ASLA, 4A OPBYT LSRA, 2A OPBYT ROLA, 6A OPBYT RORA,
 4 18 OPBYT CLC,  D8 OPBYT CLD,  58 OPBYT CLI,  B8 OPBYT CLV,
 5 CA OPBYT DEX,  88 OPBYT DEY,  E8 OPBYT INX,  C8 OPBYT INY,
 6 EA OPBYT NOP,  48 OPBYT PHA,  08 OPBYT PHP,  68 OPBYT PLA,
 7 28 OPBYT PLP,  40 OPBYT RTI,  60 OPBYT RTS,  00 OPBYT BRK,
 8 38 OPBYT SEC,  F8 OPBYT SED,  78 OPBYT SEI,  AA OPBYT TAX,
 9 A8 OPBYT TAY,  BA OPBYT TSX,  8A OPBYT TXA,  9A OPBYT TXS,
10 98 OPBYT TYA,
11 ( BUILD JUMP OPCODES                                              )
12 : JMP, 4C C, , ;
13 : JMI, 6C C, , ;
14 : JSR, 20 C, , ;
15

Screen 9     9 hex
 0 ( SK-FORTH  6502 ASSEMBLER-6                    J.W.B.  26:09:81 )
 1 : NOT, 20 XOR ;  : BEGIN, HERE 1 ;
 2 : UNTIL, SWAP 1 ?PAIRS NOT, C, HERE - 1- C, ;
 3 : AGAIN, 1 ?PAIRS JMP, ; : IF, NOT, C, HERE 0 C, 2 ;
 4 : ELSE, 2 ?PAIRS B8 C, 50 C, HERE 0 C, OVER
 5       HERE SWAP - 1- ROT C! 2 ;
 6 : THEN, 2 ?PAIRS HERE OVER - 1- SWAP C! ;
 7 : WHILE, SWAP 1 ?PAIRS IF, 2 + ;
 8 : REPEAT, 4 ?PAIRS B8 C, 50 C, HERE OVER - SWAP C!
 9          HERE - 1- C, ;
10  ( CREATE ASSEMBLER CONDITIONALS                                  )
11 30 CONSTANT 0<,  10 CONSTANT 0>,  F0 CONSTANT 0=,
12 B0 CONSTANT CS,  90 CONSTANT CC,  D0 CONSTANT 0<>,
13 : TOP 0 X, ;  : SEC 2 X, ;  : R 101 X, ;
14 : END-CODE BASE ! ?CSP SMUDGE [COMPILE] FORTH ;
15  FORTH DEFINITIONS DECIMAL
```



```
Screen 30    1E hex
 0 ( SK-FORTH EDITOR VOCABULARY-1                  J.W.B.  26:09:81 )
 1 EDITOR DEFINITIONS HEX
 2 : ETASK ;
 3 : TEXT                      ( ACCEPT THE FOLLOWING TEXT TO PAD )
 4     HERE  C/L  1+  BLANKS  WORD   PAD  C/L  1+ CMOVE ;
 5 : LINE              ( RELATIVE TO SCREEN LEAVE ADDRESS OF LINE )
 6     DUP  FFF0  AND  17  ?ERROR  SCR  @  (LINE)  DROP ;
 7 : -MOVE  ( MOVE IN BLOCK BUFFER       ADDRESS FROM-2, LINE TO-1 )
 8     LINE C/L CMOVE UPDATE ;
 9 : H                                 ( HOLD NUMBERED LINE AT PAD )
10     LINE  PAD  1+  C/L  DUP  PAD  C!  CMOVE  ;
11 : E                          ( ERASE NUMBERED LINE WITH BLANKS  )
12     LINE  C/L  BLANKS  UPDATE ;
13 : S                        ( SPREAD MAKING NUMBERED LINE BLANK   )
14     DUP  1  -  0E  DO  I  LINE  I  1+  -MOVE -1  +LOOP  E  ;
15

Screen 31    1F hex
 0 ( SK-FORTH EDITOR VOCABULARY-2                  J.W.B.  26:09:81 )
 1 : D                    ( DELETE NUMBERED LINE BUT HOLD AT PAD )
 2     DUP  H  OF  DUP  ROT  DO  I 1+  LINE  I -MOVE  LOOP  E ;
 3 : SPCR              ( DO CR IF TW <= 64 ELSE DO A SPACE       )
 4      TW  48  <  IF  CR  ELSE  SPACE  THEN ;
 5 : T                    ( TYPE NUMBERED LINE AND ALSO HOLD AT PAD )
 6     DUP  H  DUP  3  .R  SPCR   SCR @  .LINE  CR ;
 7 : L                                    ( LIST CURRENT SCREEN )
 8     SCR  @  LIST  CR  ;
 9 : R                    ( RETURN CONTENTS OF PAD TO NUMBERED LINE  )
10     PAD  1+  SWAP  -MOVE  ;
11 : P  1  TEXT  R  ;      ( PUT FOLLOWING TEXT AT NUMBERED LINE  )
12 : I  DUP  S  R  ;   ( INSERT TEXT FROM PAD AT NUMBERED LINE  )
13 : CLEAR                      ( CLEAR SCREEN SPECIFIED BY NUMBER )
14     SCR  ! 10 0 DO FORTH I EDITOR E LOOP SAVE-BUFFERS ;
15

Screen 32    20 hex
 0 ( SK-FORTH EDITOR VOCABULARY-3                  J.W.B.  26:09:81 )
 1
 2
 3 : COPY  SAVE-BUFFERS        ( DUPLICATE SCREEN-2 ONTO SCREEN-1 )
 4       B/SCR * SWAP B/SCR * B/SCR OVER + SWAP
 5       DO  DUP  FORTH  I  EDITOR  BLOCK  2- !
 6       1+  UPDATE  LOOP  DROP  SAVE-BUFFERS ;
 7
 8
 9 : FIX        ( ECHO NUMBERED LINE AND GO INTO LINE EDIT MODE   )
10     DUP  PAD  50  0 FILL LINE C/L -TRAILING  PAD  SWAP  CMOVE
11     DUP  E DUP  3 .R  SPCR  PAD  C/L  EDIT  ROT  LINE
12     SWAP  CMOVE  ;
13
14 FORTH DECIMAL
15
```

```
Screen 33    21 hex
 0 ( KTM-2 CURSOR AND CONTROL FUNCTIONS  J.W.B. MARCH 17, 1981     )
 1  FORTH DEFINITIONS HEX : KTASK ;        ( FORGET KTASK TO DUMP )
 2 : ES-EM 1B EMIT EMIT ;  ( SEND ESCAPE AND EMIT TOP STACK       )
 3 : CS 45 ES-EM BEEP ;    ( CLEAR SCREEN AND BEEP                )
 4 : HC 48 ES-EM ;         ( HOME CURSOR                          )
 5 : CES 4A ES-EM ;        ( ERASE END OF SCREEN                  )
 6 : CEL 4B ES-EM ;        ( ERASE END OF LINE                    )
 7 : BR  52 ES-EM ;        ( BEGIN REVERSE                        )
 8 : ER  72 ES-EM ;        ( END REVERSE                          )
 9 : BG  47 ES-EM ;        ( BEGIN GRAPHICS                       )
10 : EG  67 ES-EM ;        ( END GRAPHICS                         )
11 ( HOR VER CABS    0 0 CABS   HOMES CURSOR                      )
12 : CABS 3D ES-EM 20 + EMIT 20 + EMIT ;
13 ( HOR VER CREL   0 0 CREL DOES NOT MOVE                       )
14 : CREL 2B ES-EM 20 + EMIT 20 + EMIT ;
15 DECIMAL

Screen 34    22 hex
 0 ( FANCY EDITOR USING KTM-2 FEATURES       J.W.B. MARCH 17, 1981)
 1 ( ALL NEW COMMANDS TAKE TWO LETTERS                            )
 2 EDITOR DEFINITIONS HEX
 3 : BK 0 12 CABS CES ;          ( MOVE BACK TO BOTTOM OF SCREEN  )
 4 ( TITLE FOR TOP OF SCREEN                                      )
 5 : TI HC ." EDITOR ( LI,LL,PP,EE,DD,SS,II,MM,CC,<<,>> ) " BK ;
 6 : LL CS  L TI ;
 7 ( SHORT CALL TO EDITOR                                         )
 8 FORTH DEFINITIONS : ED [COMPILE] EDITOR EDITOR LL ;
 9 EDITOR DEFINITIONS HEX
10 : << -1 SCR +! LL ;  : >> 1 SCR +! LL ;
11 : LI CS LIST TI ;          ( CLEAR SCREEN THEN LIST AS BEFORE  )
12 : FL DUP 0 SWAP 2+ CABS ;          ( POSITION CURSOR AT LINE  )
13 : PP FL FIX BK ;                 ( PUT OR FIX LINE IN SCREEN  )
14 : EE DUP E FL 3 .R 40 SPACES BK ;     ( ERASE LINE ON SCREEN  )
15 DECIMAL

Screen 35    23 hex
 0 ( FANCY EDITING FUNCTIONS FOR KTM CONTINUED  JWB MARCH 17, 1981)
 1 EDITOR DEFINITIONS HEX
 2 : TT  FL 44 SPACES FL T ; ( TYPE LINE AT SCREEN POSITION )
 3 ( TYPE INDICATED LINE FROM PAD                                 )
 4 : RR DUP R FL TT ;
 5 ( SPREAD AT INDICATED LINE                                     )
 6 : SS DUP S  10 SWAP DO FORTH I EDITOR TT LOOP BK ;
 7 ( DELETE LINE AND MOVE REST DOWN                               )
 8 : DD DUP D  10 SWAP DO FORTH I EDITOR  TT LOOP BK ;
 9 ( MOVE FROM  TO INDICATED LINE ERASE FROM LINE                 )
10 : MM SWAP DUP H SWAP RR EE ;
11 ( COPY FROM TO INDICATED LINE DO NOT ERASE                     )
12 : CC SWAP H  RR BK ;
13 ( SPREAD TO ALLOW INSERTION AT INDICATED LINE                  )
14 : II DUP SS BEEP PP ; FORTH DECIMAL
15
```



```
Screen 12     C hex
 0  ( SYM-FORTH DISK COPY ROUTINE                  J.W.B. 27:04:81 )
 1 FORTH DEFINITIONS HEX
 2
 3 : BOOT FIRST USE ! FIRST PREV ! EMPTY-BUFFERS 1 WARNING ! ;
 4
 5 : CSET 7216 C! 1 7217 C! 7218 C! 6000 7210 !
 6                67FF 7212 ! MDISK ;
 7
 8 : CMESS CR ." READY TO COPY FROM 1 TO 2 ? (Y/N) " ON
 9        KEY OFF 59 = 0= 8 ?ERROR ;
10
11 : DCOPY CMESS 22 0 DO SET1 0 I CSET SET2 1 I CSET LOOP OFF ;
12 DECIMAL
13 : LOADS  OVER  +  SWAP  DO  I 12 EMIT BEEP
14          BASE  @  OVER  LIST  BASE  !  LOAD LOOP ;
15 ;S

Screen 33    21 hex
 0 ( INTERRUPT DRIVEN PADDLE DEMO-1              J.W.B.  13:05:81 )
 1 ( CONNECT JUMPER POINT CC TO CONECTION POINT P ON AA CONECTOR  )
 2 ( SEE SYM-PHYSIS 2-23 ) FORTH DEFINITIONS HEX
 3 ( IF DFLAG<>0 THEN IKEY HAS CURRENT KEY PUSHED, YOU MUST RESET )
 4 ( DFLAG TO ZERO AFTER GETTING VALUE OF IKEY                    )
 5 FORTH DEFINITIONS 79-STANDARD 0 CONSTANT IKEY  0 CONSTANT DFLAG
 6 CODE TIN   8188     JSR,   00   IM, LDA,   F9    ZP, STA,
 7           A402  AB, LDA,   8A6A     JMP,   END-CODE
 8 CODE IOUT  SEI, 8AA0 JSR,    CLI,    RTS,   END-CODE
 9 CODE IRK                PHA,  ' TIN     JSR,   7F   IM, AND,
10             ' IKEY AB, STA,     11 IM, CMP,
11          0=, IF, 02 IM, LDA,   AC0E AB, STA,   AC01 AB, LDA,
12                 A0 IM, LDA,   16C0 AB, STA,   8A   IM, LDA,
13               16C1 AB, STA,   THEN,
14             ' DFLAG AB, DEC,  AC01 AB, LDA,
15                         PLA,          RTI,   END-CODE

Screen 34    22 hex
 0 ( INTERUPT DRIVEN PADDLE DEMO-2               J.W.B. 13:05:81 )
 1 CODE INIT   SEI,   PHA,   00     IM, LDA,   ' DFLAG AB, STA,
 2       AC0C   AB,   LDA,   01     IM, ORA,     AC0C  AB, STA,
 3         82   IM,   LDA,   AC0E   AB, STA,   ' IRK   IM, LDA,
 4       A678   AB,   STA,   ' IRK 100   /              IM, LDA,
 5       A779   AB,   STA,   ' IOUT IM, LDA,     16C0  AB, STA,
 6 ' IOUT 100 / IM,   LDA,   16C1   AB, STA,     AC01  AB, LDA,
 7            PLA,   CLI,   NEXT       JMP,     END-CODE
 8 FORTH DEFINITIONS DECIMAL     ( < MOVES LEFT AND > MOVES RIGHT )
 9 VARIABLE PPOS     VARIABLE DIR    40 PPOS !    0 DIR !
10 : DPDL PPOS  @  23  CABS  BG  ." ttttttttt"  EG  ;
11 : GDIR  0  DIR  !  DFLAG  IF  IKEY  44  =  IF  -3  DIR  !  THEN
12       IKEY  46  =  IF 3 DIR  !  THEN  0  ' DFLAG  !  THEN  ;
13 : MOVPDL GDIR  DIR  @  IF  0  23  CABS  CEL   DIR  @  PPOS  @
14         +  0  MAX   69  MIN  PPOS  !  DPDL  THEN  ;
15 : DO-IT CS DPDL INIT BEGIN MOVPDL  IKEY 17 = UNTIL ;
```

```
Screen 39    27 hex
  0 ( DONALD FULL'S UTILITIES                          J.W.B. 24:04:81 )
  1 ( S<>M   M<>M     STACK-MEMORY SWAP   MEMORY-MEMORY SWAP )
  2
  3 FORTH DEFINITIONS
  4
  5 ( S<>M          N1 ADDR -> N2      SWAPS N1 WITH N2 AT ADDR )
  6
  7 : S<>M   DUP @ ROT ROT ! ;
  8
  9 ( M<>M        ADDR1 ADDR2 -> ----   SWAPS 2 VALUES IN MEMORY )
 10
 11 : M<>M   DUP @ ROT S<>M SWAP ! ;
 12 ( IN FORTH,  <>  SOMETIMES DENOTES  " NOT EQUAL TO ", BUT )
 13 ( I HAVE USED A CONVENTION OF HEWLETT-PACKARD, WHERE   <>  )
 14 ( DENOTES SOME KIND OF SWAP )
 15

Screen 40    28 hex
  0 ( DONALD FULL'S UTILITIES                          J.W.B.  24:04:81 )
  1 ( #.      PRINT N WITH ANYBASE        N ANYBASE #. -> --- )
  2 FORTH DEFINITIONS DECIMAL
  3
  4 :  #.   BASE S<>M  SWAP  .  BASE ! ;
  5
  6 ( #.R      PRINT N FORMATTED, WITH ANYBASE                  )
  7 (                          N FIELD ANYBASE  #.R -> --- )
  8 :  #.R   BASE S<>M  ROT ROT  .R  BASE ! ;
  9
 10 : B.     2 #. ;         ( BINARY PRINT          N B. -> --- )
 11 : O.     8 #. ;         ( OCTAL PRINT           N O. -> --- )
 12 : H.    16 #. ;         ( HEX PRINT             N H. -> --- )
 13 : B.R    2 #.R ;  ( BINARY PRINT FMTD   N FIELD B.R -> --- )
 14 : O.R    8 #.R ;  ( OCTAL PRINT FMTD    N FIELD O.R -> --- )
 15 : H.R   16 #.R ;  ( HEX PRINT FMTD      N FIELD H.R -> --- )

Screen 41    29 hex
  0 ( DONALD FULL'S UTILITIES                          J.W.B.  24:04:81 )
  1
  2 FORTH DEFINITIONS DECIMAL
  3
  4 : VERIFY      (  <addr> VERIFY -> ----                              )
  5           DUP CR 0 6 D.R DUP 8 + SWAP DO I C@ 4 H.R LOOP SPACE ;
  6 : DUMP      ( <addr1> <addr2> DUMP -> ---          MEMORY DUMP )
  7         SWAP DO I VERIFY 8 +LOOP SPACE ;
  8
  9 : .HS 1 HARD ! ;  : .HC 0 HARD ! ; ( HARD COPY SET AND CLEAR    )
 10 : HLIST .HS CR CR LIST .HC ;       ( HARD COPY LIST COMMAND    )
 11 : HTRIAD .HS CR TRIAD .HC ;        ( HARD COPY TRIAD COMMAND   )
 12 ( <last+1> <first>  RPUT or RGET -> ----                        )
 13 : RPUT DO I I PUT CR I I . . ." PUT " LOOP ;  ( RANGE PUT       )
 14 : RGET DO I SGET CR I . ." SGET " LOOP ;      ( RANGE SGET      )
 15
```


A SMALL MUSIC LANGUAGE USING CB2 SOUND

```
Screen 6     6 hex
  0 ( STEP 1 - DEFINE THE SOFTWARE INTERFACE TO THE HARDWARE       )
  1
  2 FORTH DEFINITIONS HEX
  3
  4 A80B CONSTANT CONTROL.REGISTER   A808 CONSTANT FREQUENCY
  5
  6 A80A CONSTANT SHIFT.REGISTER
  7
  8 DECIMAL
  9
 10 : ON  16  CONTROL.REGISTER C!  0 FREQUENCY C! ;
 11
 12 : OFF 00  CONTROL.REGISTER C! ;
 13
 14 : TONE  ON  15 SHIFT.REGISTER C!   FREQUENCY C! ;
 15


Screen 7     7 hex
  0 ( STEP 2  CREATE THE APPLICATION LANGUAGE                       )
  1
  2   VARIABLE TIME.BASE    500 TIME.BASE !
  3 : 1BEAT TIME.BASE @ 0 DO I DROP LOOP ;
  4 : NOTE CREATE , DOES> @ TONE 0 DO 1BEAT LOOP OFF ;
  5
  6 240 NOTE C   226 NOTE C#  213 NOTE D   201 NOTE D#  190 NOTE E
  7 179 NOTE F   169 NOTE F#  160 NOTE G   151 NOTE G#  142 NOTE A
  8 134 NOTE A#  127 NOTE B   119 NOTE C2  113 NOTE C2# 106 NOTE D2
  9 100 NOTE D2#  95 NOTE E2   89 NOTE F2   84 NOTE F2#  80 NOTE G2
 10  75 NOTE G2#  71 NOTE A2   67 NOTE A2#  63 NOTE B2   59 NOTE C3
 11  56 NOTE C3#  53 NOTE D3   50 NOTE D3#  47 NOTE E3   44 NOTE F3
 12  42 NOTE F3#  40 NOTE G3   37 NOTE G3#  35 NOTE A3   33 NOTE A3#
 13               0  NOTE R    ( REST )
 14 : SCALE 1 C 1 D 1 E 1 F 1 G 1 A 1 B 2 C2 2 C2
 15         1 B 1 A 1 G 1 F 1 E 1 D 4 C ;


Screen 8     8 hex
  0 ( STEP 3  WRITE THE APPLICATION PROGRAM                         )
  1 ( IN THIS CASE IT IS A SONG CALLED " TURKEY "                   )
  2 ( THE SONG HAS 6 PARTS SOME OF WHICH ARE REPEATED               )
  3 : P1     1 F2# 1 E2  1 D2  1 C2# 1 D2  1 E2  1 D2  1 A 1 F# 1 G
  4          1 A   1 B   1 A   1 F#  2 A   1 D2  1 E2 ;
  5 : P2     2 F2# 2 F2# 1 F2# 1 E2  1 D2  1 E2  1 F2#  1 E2 1 D2
  6          1 F2# 2 E2  ;
  7 : P3     1 F2# 1 E2  1 F2# 1 G2  1 A2  1 F2# 1 D2  1 E2
  8          1 F2# 1 D2  1 E2  1 A   2 D2  2 R ;
  9 : P4     1 F#  1 E   1 F#  1 G   1 A   1 G   1 F#
 10          1 E   1 F#  1 E   1 F#  1 G ;
 11 : P5     1 D2  1 E2  1 F2# 1 D2  1 B   1 C2# 1 D2  1 A
 12          1 F#  1 G   1 A   1 F#  2 E   1 D   1 E ;
 13 : P6     1 F#  1 E   1 F#  1 G   1 A   1 F#  1 D
 14          1 E   1 F#  1 D   1 E   1 C#  2 D   2 R ;
 15 : TURKEY P1  P2  P1  P3  P4 2 A 2 R P4 2 B 1 B 1 C2# P5 P6 ;
```

THIS PAGE RESERVED FOR YOUR NOTES