# MICRO MANUAL
for SUPER-FORTH 64™
By Bruce Jordan

INTRODUCTION

The object of the MICRO MANUAL is to get the person that has little prior knowledge of FORTH using the SUPER FORTH system as quickly as possible. If you have no prior knowledge of FORTH, we strongly suggest that you get a copy of STARTING FORTH, by Leo Brodie. STARTING FORTH is an excellent first book on the subject, and can be ordered directly from PARSEC RESEARCH.

ABOUT FORTH

The MICRO MANUAL assumes that your programming experience has been largely that of BASIC and machine language. So you may be asking yourself: "What´s so different about FORTH, aside from the fact that it runs faster than BASIC?" The answer is: FORTH is like no other programming language you´ve ever used!

Think for a moment about how you program in BASIC. You usually start with an idea about what you want your program to do. Then you try to get the big picture of how the program is going to accomplish its task. Finally, you start writing the code, which means writing a long series of numbered lines of statements that, depending on the complexity, interconnect and intertwine into a huge maze of hopefully correct instructions. However, if you´re one of the mortals, you know as well as I that getting it right the first time is seldom the case; and tracking down a bug in a large BASIC program can range anywhere from unpleasant to down right nightmarish. Programming in machine language is about the same, only worse. But why should this be the case?

Much of the problem with BASIC and machine language programming lies in their unstructured nature. When programming in BASIC, people have a tendency to just start typing instructions until they reach what they think is the end. BASIC supports this sort of programming, even though it´s not very efficient. Think about the last BASIC program you wrote. It probably contained several IF/THEN statements linked to a whole slew of GOTO commands that were created as needed, or because you forgot to include something. These GOTO commands probably ran here and there all through the program. Sure, GOTO commands are convenient, but at the same time, it´s real easy to cut your own throat with them. In fact, most of the problems programmers have with BASIC can be traced directly to GOTO commands.

BASIC programs must be taken as a total when de-bugging. By this, I mean that when you run your BASIC program for the first time, and it doesn´t work properly, you know that the error is in there somewhere, but finding exactly where is usually a matter of guess work. Executing just one small part of a BASIC program to look for a suspected bug is almost impossible if that section contains several GOTO and GOSUB commands. And if you do find the problem, you don´t have any guarantee that the change you´ve made is not going to cause something else in your program to go awry. However, FORTH was designed with just these problems in mind.

FORTH is a STRUCTURED PROGRAMMING LANGUAGE.

In essence, what this means is: NO GOTO STATEMENTS. A program written in FORTH has a rather elegant, linear quality to it. Each command follows the last in a one-command-at-a-time fashion, yet FORTH´s structure isn´t so linear that it cramps your style. By the use of different specialized LOOPing commands, FORTH becomes as flexible, if not more flexible, than BASIC. By the use of loops and linear programming, program code becomes much more straight-forward and readable. But the real power of FORTH, the key to its structured nature and ease of de-bugging lies in its modular nature.

FORTH is a MODULAR PROGRAMMING LANGUAGE.

You program in FORTH by creating small, simple modules that perform specific tasks. Then you string these modules together to create your program. It´s like making a program out of a whole bunch of little programs; and each of the little programs (modules) may be executed by themselves for the purpose of testing. For instance, below is one of these modules. When executed, it will print HI THERE!.

```
: HELLO ." HI THERE!" ;
```

Every module created in FORTH has three things in common. First, the module starts with a COLON. Second, a module must have a name. In the above example, the name is "HELLO". Finally, a module must end with a SEMICOLON to mark the end of the module. Everything else between the NAME and the SEMICOLON is your code. For instance, in the above example ." (Pronounced DOT QUOTATION) was used. DOT QUOTATION tells your computer to print everything following the quotation mark until it finds the ending quotation mark. In this case it prints HI THERE!.

In FORTH, modules are called WORDS. When FORTH WORDS are created they´re COMPILED into a collection of WORDS in the computer´s memory. This collection of FORTH WORDS is called the DICTIONARY. WORDS can even be linked together to form new WORDS and, in turn, programs. What you wind up with is highly structured code that´s more compact, runs more efficiently and faster than an equivalent BASIC program.

Here´s an example of linking WORDS together:

```
: HELLO ." HI THERE!" ; ( OUR FIRST MODULE)
: GREET ." WELCOME TO SUPER FORTH" ; ( PRINTS WELCOME TO SUPER)
                                     ( FORTH ON THE SCREEN)
```

THEN WE CAN LINK THESE TWO MODULES TOGETHER:

```
: SAYHI HELLO GREET ;
```

Now, anytime we execute SAYHI, by typing SAYHI followed by a CARRIAGE RETURN (Or by having another word execute it.), we get HI THERE! WELCOME TO SUPER FORTH! printed on the screen. As another example, we could use our two modules like this:

```
: 10HI 10 0 DO SAYHI LOOP ; ( REMEMBER: SAYHI IS ACTUALLY TWO WORDS.)
```

The module 10HI will print HI THERE! WELCOME TO SUPER FORTH! on the screen ten times.

With BASIC, you have a certain number of commands available, and that´s all. However, notice that with FORTH, each little WORD you create can be used in either the program <u>or</u> the immediate mode. What this means is each WORD once created acts as a brand new command! FORTH is EXTENSIBLE. You can keep adding new commands to your system as long as available memory holds out!

So what does this mean in terms of programming ease? It means that since our program is a series of smaller, stand-alone WORDS, de-bugging a program is a darn sight less painful. Each WORD can be easily tested to see if it´s the culprit. And if a WORD needs to be changed, it can be without having to re-write the entire program. But this is only the beginning.

So far, we´ve said that FORTH is: A high-level language, it executes fast, its modular form makes it easier to program with and de-bug, and its also extendable. So, what makes SUPER FORTH SUPER? SUPER FORTH takes the concepts of FORTH programming ease one step further by supplying, already written, many of the of more complicated modules that you would have to write yourself! This saves you from having to re-invent the wheel, and gets you closer to getting your job done. For instance, you probably know that your COMMODORE 64 is a powerful graphics machine, with hi-res bitmapping, multi-color capabilities, sprites, the works! But you also know that the 64´s BASIC is completely devoid of graphics commands, and if you´ve ever tried doing graphics programming, you know how painful that can be! SUPER FORTH to the rescue! SUPER FORTH is loaded with graphic commands all ready to go. You can set up hi-res screens, split-screens, draw lines, circles, ellipses; change colors, erase, just about anything you can think of. The same is true for music, interrupts, string handling, I/O applications and more. SUPER FORTH makes your 64 talk your language, instead of you mumbling its! But, enough about what SUPER FORTH can do, let´s get started so YOU can start DOING.

GETTING STARTED

NOTE: Whenever you see: <CR> this means for you to hit the RETURN key.

The first thing you want to do is to make a working copy of your SUPER-FORTH disk. A working copy differs from a backup copy in that there will be no FORTH screens on your working copy, as there are on your Master disk (The master disk is the one you purchased form PARSEC RESEARCH.). If you don´t know what FORTH screens are, don´t worry, they will be discussed later. If you wish to make a backup copy of your Master disk, instead of a working copy, follow the instructions for BACKUP in the Backup Utilities section in the main manual.

MAKING A WORKING COPY

1. Turn on your computer and disk-drive.
2. Insert your SUPER FORTH disk into your disk-drive, and type: LOAD"SUPER FORTH 64",8 and hit RETURN.
3. When your computer is finished loading, type RUN and hit RETURN. The message: SUPER FORTH 64 VERSION... will appear on the screen.
4. Remove your SUPER FORTH disk, and put it away in a safe place.
5. Insert a new, formatted disk into your disk-drive, and type: SAVE-FORTH. Your disk-drive will start running. When it stops, you now have your working copy of SUPER FORTH.

NOTE : SAVE-FORTH does not copy the SUPER FORTH extension screens from the master disk to your working disk. These screens have already been compiled into the system, and are included only for re-configuring SUPER FORTH.

THE STACK

By far, the most important aspect of FORTH is the STACK. The STACK is what makes FORTH what it is, and is the source of the most confusion for BASIC programmers.

Just about everything that FORTH does is centered around the STACK. Consider a mathematical operation in BASIC, say, adding two numbers, and then multiplying by a third number:

```
10 A=5:B=6:C=2
20 D=A+B*C
30 PRINT D

RUN
 17

READY.
```

Notice that in BASIC, we assign values to variables, then we perform mathematical operations on the variables. With FORTH, however, when we perform these sorts of operations, we use an area in memory called the STACK. First, we put the values on the STACK, either directly, or by reading in the contents of a memory location. Then we perform our operations on the numbers. This is really a lot simpler than it sounds. The main thing to remember is that most operations in FORTH occur on the STACK.

The STACK has a FIRST-IN-LAST-OUT structure. What this means is if we enter three numbers, say 3 4 and 5, since the 5 was entered last, it will be the number on top of the STACK, with 4 next and 3 at the bottom of the STACK.

EXAMPLE :

The . "period" (PRONOUNCED DOT), is the FORTH word for printing out a number on the STACK. Lets enter our three numbers now...

```
3 4 5 <CR> OK
```

We now have three numbers sitting on the STACK. We can see these numbers by printing them using the DOT command:

```
. <CR> 5 OK
. <CR> 4 OK
. <CR> 3 OK ( LETS DO DOT ONE MORE TIME.)
. <CR>
^STACK EMPTY (NOTHING ELSE ON THE STACK)
```

Notice that 5 was the first number out, though it was the last entered, and 3 was the last number out, though it was the first entered. This is what is meant by FIRST-IN-LAST-OUT structure. This leads to a rather unfamiliar way of doing math, but it works out well once you get used to it. For instance, let´s use FORTH to do the same math operation that we did in the BASIC example:

```
5 6 + 2 * <CR> OK ( THE RESULT IS NOW ON THE STACK AND THE REST)
                  ( OF THE NUMBERS ARE GONE.)
```

If we print out our result, we get...

```
.<CR> 17 OK
```

Notice that with FORTH, the operator (+ - * /) appears after a pair of numbers, and <u>not</u> between them. Let´s try an example using subtraction:

```
17 5 - <CR> OK
. <CR> 12 OK
```

Once again, the number we subtracted from went on the stack first, followed by the number we subtracted from it, followed by the operator. As a general rule, all FORTH words take the operand first, followed by the operator.

One of the mistakes that beginning FORTH users frequently make in connection with the STACK is the over use of variables. In BASIC, almost everything is assigned a variable. However, good FORTH programming technique requires that the use of variables be kept to a minimum. The key to this is learning how to use the FORTH STACK MANIPULATION words. The beginner will place a couple of values on the STACK, do something with them, then place a third value on the STACK. Then he or she may say: "Oh Heck! (Or something like that) The value on the bottom of the STACK is the one I want to be on top." So, he or she will create a couple of variables to temporarily hold some of the STACK values so that they can get the STACK in the order needed.

However, by the use of the STACK MANIPULATION words, large numbers of values can be kept on the STACK, and easily moved about in any way desired. Variables, when ever possible, should be used only for data.

FORTH STACK MANIPULATION WORDS

SWAP	Swaps the two top numbers on the stack.
OVER	Copies the second value on the stack to the top of the stack.
ROT	Rotates the third value on the stack to the top of the stack.
ROLL	Moves the nth value on the stack on to the top of the stack.
PICK	Copies the nth value on the stack to the top of the stack.
DUP	Duplicates the top number on the stack.

By gaining a good understanding of how to use these words, your programs will be more readable, will execute faster and will be more compact.

THE COLON DEFINITION

The next thing you should know is how to create a FORTH COLON DEFINITION.

All COLON DEFINITIONS have two things in common: they always begin with a COLON, and they almost always end with a SEMI-COLON. The COLON tells SUPER FORTH that what follows is to be compiled into the system, and the SEMI-COLON signals the end of the definition.

EXAMPLE :

```
: HELLO  ." HI THERE! "  ;  ( THIS IS A COLON DEFINITION)
```

Notice the COLON at the beginning of the line, and the SEMI-COLON at the end. Type the above line, including all spaces, followed by a carriage return. Now, type HELLO and hit RETURN. If you´ve done everything correctly...

```
HI THERE! OK
```

should appear on the screen. Congratulations! you´ve just created your first FORTH word. By the way, the "OK" is SUPER FORTH´s way of telling you that it´s finished. OK is FORTH´s equivalent of "READY" in BASIC. Also, note that unlike COMMODORE BASIC, FORTH commands must be separated by spaces. Also, the entire definition need not be on the same line:

EXAMPLE :

```
: TEST2 ." THIS IS ON ONE LINE"  <CR>
10 0 DO  <CR>
65 EMIT  <CR>
LOOP ;   <CR> OK
```

The above example will work fine, even though parts of the definition are on different lines. Just make sure to hit a carriage return at the end of each line, and end the definition with a semi-colon.

Now all that remains is to start creating your own COLON DEFINITIONS by combining different SUPER FORTH commands.

BUT REMEMBER

1. ALL COLON DEFINITIONS MUST START WITH A COLON AND END WITH A SEMI-COLON.

2. <u>FORTH COMMANDS MUST BE SEPARATED BY SPACES.</u>

USING FORTH SCREENS

The next question is: "If I´m just making up these little words all over the place, how will I keep track of them long enough to make a program of any size?" The answer is: FORTH Editor Screens.

Try this: format a blank disk.* You don´t want any other sequential or program files on the same disk that will contains FORTH Editor Screens (More about this later.). Next, put it into your disk-drive and type 1 LIST followed by a carriage return. You will see something like this appear on your screen:

```
SCREEN #1

 0)
 1)
 2)
 3)
 4)
 5)
 6)
 7)
 8)
 9)
10)
11)
12)
13)
14)
15)
OK
```

This is a blank FORTH EDITOR SCREEN. Note that you have 16 (0-15) lines to use for entering code, and each line can hold up to 64 characters. However, the number of lines to a screen, and the number of characters per line can be changed if desired. For more on this, consult the section on editor words in the main manual.

You use these EDITOR screens to create larger programs. To see some examples of this look in the back of your main manual. There, you will find many of the FORTH listings for the various SUPER FORTH extensions. These screens have already been compiled into the system, and are provided for reference, and to allow you to re-configure your SUPER FORTH. Another place where

* See pg3 of SF64 manual- " NØ:diskname,id" DOS

these FORTH Extensions are listed is on your Master disk. To see them, insert your Master disk, type the number of the screen you wish to see, and type LIST.

Screens allow you to make larger programs containing many COLON DEFINITIONS. To enter code, use the cursor keys to get to a particular line number, just as you would with BASIC. Type in your code and then hit return. Make sure to leave a space between the line numbers and your code.

EXAMPLE :

TYPE...

```
1 LIST  <CR>

W <CR>
```

* W "WIPES" the screen clear. ALWAYS DO THIS! Even if the screen looks empty, there may be non-printing characters hiding on the screen that can really screw things up.

Next, fill in screen 1 as shown below. Remember the carriage return at the end of the lines.

SCREEN1

```
 0) : TEST 10 0 DO ." THIS IS A SCREEN TEST"
 1) LOOP ;
 2)
 3) : TEST2 10 0 DO ." SUPER FORTH"
 4) TEST LOOP ;
 5) ( THIS IS A COMMENT)
 6) ( THIS IS ANOTHER COMMENT)
 7)
 8)
 9)
10)
11)
12)
13)
14)
15)
OK
```

Note the comments contained within the parentheses. The parentheses allow you to add comments to your screens, but unlike BASIC, FORTH comments are not compiled into the system, and do not slow down execution.

* Note, also, that there must be one space between the first parentheses and the start of your comment.

Now, move the cursor off the FORTH EDITOR SCREEN and type:

F <CR>

You will hear your disk-drive start. What you have just done is created a FORTH screen and saved it to the disk. To show you what you just did, type:

EMPTY-BUFFERS <CR>

This will clear out any screens held in memory, including the one you just created. Now type 1 LIST. You will hear your drive start again, and the screen you just created will appear. As you see, your FORTH screen is on the disk. If for some reason you want to change it, simply make your changes, and type F once again, and your new version of the screen will be saved in place of the old version. It´s as easy as that.

With the default SUPER FORTH system, as many as eight EDITOR screens can be kept in memory at one time. However, by using the FORTH command: #BUFF you can change the number of EDITOR screens that the system can hold. See the main manual for more on this.

There are many powerful editing commands contained in SUPER FORTH. Refer to the Editor section in your main manual for more on these.

LOADING FORTH EDITOR SCREENS

The next question is: When we get our FORTH code onto screens, how do we turn our screens into working programs? We do this by LOADING the screens. There are two words for doing this:

LOAD and THRU.

LOAD loads a single screen, or a series of screens strung together by the FORTH command "-->". THRU is used to load a series of separate screens.

For example, if we have just one screen we wish to load, like the one we just created, then we use LOAD. Try this. Type the screen number, followed by LOAD:

```
1 LOAD <CR>
```

Now, type: TEST2 <CR>

A whole bunch of printing should appear on your screen. What we just did was to load the colon definitions contained on screen #1 into the SUPER FORTH system.

Another time when we use LOAD is when we don´t have enough room on a particular screen to finish a colon definition:

ONLY THE LAST FEW LINES OF THE EDITOR SCREEN ARE SHOWN.

```
12)
13)
14) : TEST3 10 0 DO TEST2 PAGE
15) ." THIS IS A BLANK SCREEN" 5 BKGND ( NO ROOM TO FINISH)
OK
```

The "-->" command allows us to continue the definition on the next screen:

```
13)
14) : TEST 3 10 0 DO TEST2 PAGE
15) -->
OK

SCREEN #2

 0) ." THIS IS A BLANK SCREEN" 5 BKGND
 1) 5 EMIT ." NOT ANYMORE!"
 2) LOOP ;
 3)
 4)
```

✱ Screens linked in this way are ALWAYS loaded using LOAD.

Now, let´s say that we have a program made up of a series of colon definitions that take up several screens, and there are no colon definitions that cross over to the next EDITOR screen as in the last example. Then we can use the word THRU for loading the screens.

Say that our program took up screens 5-16. To load them, you would type the starting number of the EDITOR screens to be loaded, followed by the ending number of the EDITOR screens to be loaded, followed by the command THRU:

```
5 16 THRU
```

This would load all of the screens from screen 5 to screen 16.

Once your FORTH screens are loaded, your program will execute just as any other FORTH word, by typing it´s name. However, now you have several choices about how your program is used and kept. Since your FORTH screens are saved on disk, you can call them up each time you restart SUPER FORTH and load them back into the system, or you can use the FORTH command SAVE-FORTH to create another working copy of your SUPER FORTH system, but this working copy will contain your program already loaded into the system, and ready to run. Yet another choice is to use the SUPER FORTH command "APPLICATION" to create a stand-alone program that will run on any COMMODORE 64, with or without SUPER FORTH! APPLICATION makes the SUPER FORTH system "invisible" to anyone using your program, and therefore, allows you to write programs using SUPER FORTH that you may market freely, without paying license fees. For more on this, consult the main manual.

THINGS TO REMEMBER ABOUT SCREENS

1. NEVER PUT FORTH SCREENS ON A DISK WITH ANYTHING ELSE!

   Screens use Relative files, and can write over other files on the disk, and visa versa. Always keep your screens on a separate disk. This is why your working copy was made without the SUPER FORTH Extension Screens.

2. EACH LINE OF A SCREEN HAS A LENGTH OF SIXTY FOUR CHARACTERS.

3. ALWAYS WIPE EDITOR SCREENS BEFORE USING!

4. ALWAYS HIT A CARRIAGE RETURN AT THE END OF THE LINE.

5. USE LOAD WHEN COMPILING A SINGLE FORTH SCREEN, OR A SERIES OF FORTH SCREENS LINKED TOGETHER WITH "-->".

6. USE THRU FOR COMPILING A SERIES OF SELF-CONTAINED FORTH SCREENS.

7. NEVER USE "-->" WITH THRU

8. <u>DON´T FORGET TO USE "F" TO SAVE EDITOR SCREENS WHEN FINISHED EDITING!</u>

9. Screen #0 is reserved for documentation purposes only. Never put part of a program on this screen. <u>IT WILL NOT LOAD!</u>

LIST OF EDITOR COMMANDS

COPY	Copy a screen.
EDITOR	Enter EDITOR mode.
FLUSH	Flush a screen to the disk. or F
L	List current editor screen.
N L	List next editor screen.
P L	List previous editor screen.
LIST	List screen, and enter EDITOR mode.
K	Kill a line.
M	Move a line.
O	Open a line.
SC	Copy line fro different screen.
SM	Move line from different screen.
W	Wipe a screen clean.
X	Extract a line.

STRUCTURED TOP-DOWN/BOTTOM-UP PROGRAMMING

A unique feature of FORTH is its ability to allow you to do TOP-DOWN/BOTTOM-UP PROGRAMMING; and this means a savings of time and effort between the conception of a program and its completion. Here´s how it works. Say you want your computer to do something: a video game, a spread sheet, walk your dog, or whatever. In TOP-DOWN/BOTTOM-UP PROGRAMMING, you start by defining the widest possible task first, then you break down that task into smaller and smaller tasks until you reach the smallest increment. Then it´s just a matter of writing a WORD for each one of the smallest tasks, and combining these WORDS into HIGHER-LEVEL WORDS, and those WORDS into yet HIGHER-LEVEL WORDS until you reach the top (where you started from) as one single WORD.

EXAMPLE :

As a simple example of TOP-DOWN/BOTTOM UP programming, we´re going to create a word that will draw a box anywhere on a hi-res screen, and then fill the box in to form a square.

We start by choosing the highest level word. We´ll call it:

SQUARE *

Next, we think about what we´ll need to get SQUARE to work. Well, first, I said we would make a "box" and then fill it in, so SQUARE must contain the two words:

BOX and S-FILL

BOX, will make the box on the screen, and S-FILL will fill in the box to make a square. But now we need to think about what will go into these two words. Let´s take BOX first.

We have to start drawing from somewhere, so we need to mark the start. Also, a box needs sides. Therefore, we need four words that will draw the sides of the box. We´ll call our words:

START TOP RIGHTSIDE BOTTOM and LEFTSIDE

*** Your programming should begin by writing on paper.**

Knowing that SUPER FORTH has good graphics commands, we reason that we can probably define these four words using available SUPER FORTH commands. So, our design of this part of the program is finished. Now for S-FILL.

Since a square is a very regular shape, we reason that it can easily be filled in with a bunch of straight lines, and since we can use a simple DO loop to draw any number of lines we choose, we reason that S-FILL can be written as a single FORTH word. This finishes the design phase of our command SQUARE. Now all that remains is to write the code for the various words and then string them together. Keep in mind we´ll be doing this from the BOTTOM-UP. Here´s how:

First, using a newly formatted disk, we bring up a FORTH EDITOR SCREEN, by typing 10 LIST.

Next, we WIPE the screen clear, by typing W.

Now, we enter our code:

(It´s unnecessary to enter comments)

```
SCREEN #10
( TOP-DOWN/BOTTOM-UP PROGRAMMING DEMO)

: START ( X Y --- X Y )
  2DUP B-PLOT ;

: TOP ( X Y --- X Y )
  SWAP 50 + SWAP 2DUP B-LINE ;

: RIGHTSIDE ( X Y --- X Y )
  50 + 2DUP B-LINE ;

: BOTTOM ( X Y --- X Y )
  SWAP 50 - SWAP 2DUP B-LINE ;

: LEFTSIDE ( X Y --- X Y )
  50 - 2DUP B-LINE ;
```

We now have the words for drawing the various sides of the box. We will move on to writing the code for S-FILL.

We need another FORTH EDITOR SCREEN, so type N L. This will bring up SCREEN #11. Now type W to WIPE the SCREEN.

We said S-FILL will fill in the box by drawing a bunch of horizontal lines across the box. Then what we want is a word that will draw a line form one side of the box to the other, and then repeat this action from the top of the box to the bottom. Since we have the starting point on the stack (Left there by the word BOX.), and since we can use the INDEX of a DO loop, S-FILL should be a fairly straight-forward word to write.

By the way, don´t worry if you don´t understand the code. We´re doing this only to learn the concept of TOP-DOWN/BOTTOM-UP programming.

Here´s the code for S-FILL:

```
SCREEN #11
( TOP-DOWN/BOTTOM-UP PROGRAMMING DEMO)

: S-FILL ( X Y --- )
  DUP ( SAVE A COPY OF Y )
  50 + ( ADD 50 TO Y: THAT´S HOW TALL OUR BOX IS)
  SWAP DO
         DUP 50 + SWAP
         DUP I B-PLOT ( GET INDEX OF LOOP/SET STARTING POINT )
         SWAP I B-LINE ( DRAW LINE )
       LOOP
 DROP ; ( GET RID OF ANYTHING ON THE STACK )
```

Now, we´ll get one more SCREEN to define our words of the next level, and then the final level: Type N L, then W.

We can now enter the code for the next level of words:

```
SCREEN #12
( TOP-DOWN BOTTOM-UP PROGRAMMING DEMO )

( BOX LEAVES SOMETHING ON THE STACK FOR S-FILL TO USE )

: BOX ( X Y --- X Y )
START TOP RIGHTSIDE BOTTOM LEFTSIDE ;

( NOW COMES OUR HIGHEST LEVEL WORD )

: SQUARE ( X Y --- X Y  DRAW A SQUARE AT X,Y )
  BOX S-FILL ;
```

```
12)
13)
14)
15)
```

Finally, type F to FLUSH this last SCREEN to the disk. We can now LOAD our newly created words and see if they work.

Type:

```
10 12 THRU
```

Your computer should pause for a moment, while it´s compiling the words, then print the message OK. If something goes wrong, re-list your EDITOR SCREENS and look for errors.

Let´s test SQUARE:

Type 20 D-SPLIT 7 BITMAP 0 20 D-POSITION

This will give you a split graphics screen. Now type:

```
100 100 SQUARE
```

and a square should appear on the screen. Try it a few more times:

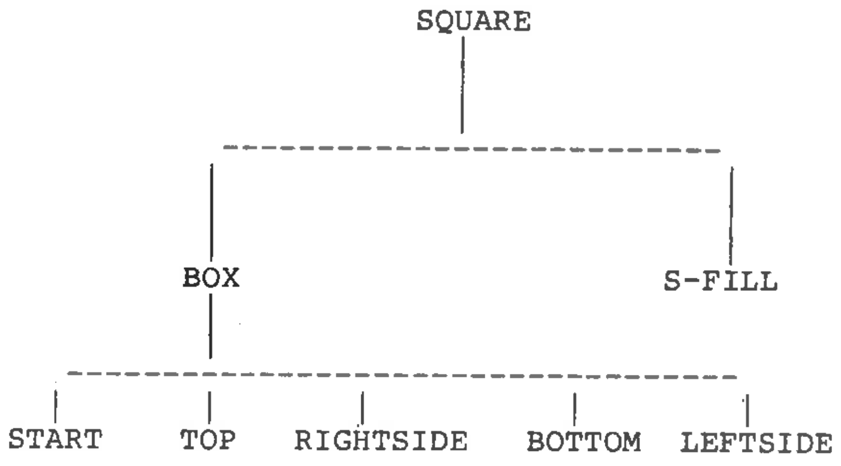
```
45 50 SQUARE
150 10 SQUARE
0 0 SQUARE
```

When you´re finished, type: 0 BITMAP 0 D-SPLIT.

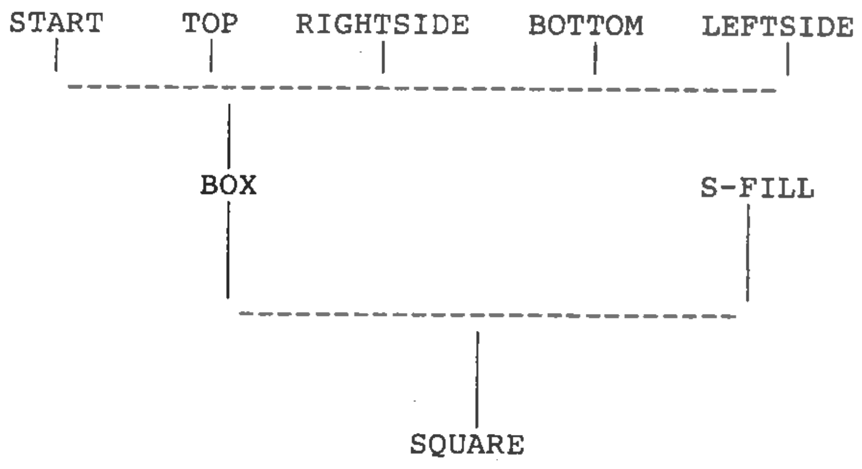
Congratulations! You´ve just successfully completed your first STRUCTURED, TOP-DOWN/BOTTOM UP program.

REMEMBER!!!

We designed our program from the TOP-DOWN, like this:



But we wrote the program from the BOTTOM-UP, like this:



The fact that each FORTH word defined is itself a mini-program, and that FORTH words can be used in the defining of other FORTH words, allows us to program in this TOP-DOWN/BOTTOM UP fashion. And if the program doesn´t work for some reason, FORTH´s modular nature allows us to test each of the modules until we find out which one we need to change. Also, if we need to add something (We usually do.), this can be done by creating another FORTH word that can be easily included into the program, without screwing everything else up.

Once you get the hang of creating and writing your programs in this fashion you´ll find that your productivity will increase at least three fold! That´s a fact.

SUPER FORTH / STARTING FORTH DIFFERENCES

1. **?STACK :**

The book
STARTING FORTH uses ?STACK for a different purpose. Starting FORTH´s use is to test for an underflow condition. While SUPER FORTH´s uses ?STACK to test if the stack is out of bounds.

2. **TICK :**

There are two differences between TICK " ´ " in STARTING FORTH and SUPER FORTH:

First, the STARTING FORTH version of TICK returns the CODE FIELD ADDRESS (CFA) of a word, while SUPER FORTH´s version of TICK returns the PARAMETER FIELD ADDRESS (PFA) of a word. To obtain the Code Field Address of a word using TICK, use TICK followed by the command CFA.

EXAMPLE :

```
´ EMIT CFA <CR> OK ( LEAVES THE CODE FIELD ADDRESS ON THE STACK)
```

Second, STARTING FORTH´s version of TICK can be used within a colon definition to compile the Code Field Address (CFA) of the next word in the input stream. However, using TICK in a SUPER FORTH (79-STANDARD) colon definition will cause the Parameter Field Address (PFA) of the next word in the definition to be compiled into the definition.

EXAMPLE :

```
: TEST ´ DUP ;
```

In the above colon definition, TICK will compile the PFA of DUP into the definition of TEST.

To compile the next word in the input stream (as STARTING FORTH´s version of TICK does), use [COMPILE] ´.

EXAMPLE :

```
: HELLO ." HI THERE " ;

: TEST [COMPILE] ´ CFA EXECUTE ; ( COMPILES CFA OF A WORD AND)
                                 ( EXECUTES IT.)
```

```
TEST HELLO <CR> HI THERE OK
```

3. **LITERAL :**

The word literal does not work the same as the example on page 304 of STARTING FORTH. SUPER FORTH´s version of LITERAL must have the values generated within the definition using the left and right square brackets ([ ]).

EXAMPLE :

```
        : TEST [ 123 ] LITERAL ;

          TEST 123 OK
```

However, there is a way to take values directly from the stack, by generating a duplicate of the stack value while compiling:

```
        HERE 10 ALLOT ( CREATE A 10 BYTE AREA IN MEMORY )
        : LIMITS 2* [ DUP ] LITERAL + ; DROP
```

The DUP in square brackets makes a copy of the top value on the STACK that LITERAL can use.

4. **S0 :**

The explanation of S0 on page 247 of STARTING FORTH says that it points to the bottom of the parameter stack <u>and</u> the address of the input buffer. In SUPER FORTH, S0 points <u>**ONLY**</u> to the bottom of the stack. <u>**TIB**</u> points to the input buffer.

5. **TEXT :**

STARTING FORTH´s version of TEXT differs from SUPER FORTH´S. SUPER FORTH´s version puts the character count at the beginning of the string, while STARTING FORTH´S version uses no character count. Because of this, the example of GREET on page 274 of STARTING FORTH will not work correctly. To get this example to work, it should read:

```
: GREET ." HELLO " MY-NAME COUNT TYPE
." , I SPEAK FORTH." ;
```

Similarly, the example of GREET on page 276 has two differences:

First, it uses TEXT in the same conflicting way as in the example above, and Second; the example on page 276 uses S0 for the input buffer pointer. This should be changed to TIB (Top of Input Buffer) for the SUPER FORTH system.

The example on page 276 should read:

```
: GREET CR ." WHAT´S YOUR NAME?" TIB @ 40 EXPECT
0 >IN ! 1 TEXT CR ." HELLO,"
PAD COUNT TYPE ." , I SPEAK FORTH."
```

6. PLUS :

The example for PLUS on page 277 doesn´t work because STARTING FORTH´s definition for NUMBER is different than SUPER FORTH´s

7. SIGN :

There is a difference in how STARTING FORTH and SUPER FORTH use the word SIGN.

The example on page 171 of Starting FORTH, shows the definition of .$ as follows:

```
: .$  SWAP OVER DABS
<# # # 46 HOLD #S SIGN 36 HOLD #> TYPE SPACE ;
```

This definition supposedly allows you to input a number such as 123.23, and then by calling $. the number will be printed in the format: $123.23.

However, if the number you enter is negative, eg: -123.23, .$ will not print the negative sign.

In order to get the negative sign, a ROT should precede SIGN. For instance, for the above example to work with SUPER FORTH, it should be in the form:

```
: .$  SWAP OVER DABS
<# # # 46 HOLD #S 36 HOLD ROT SIGN #> TYPE SPACE ;
```

Now, when 123.23 is entered, .$ returns $123.23, and if -123.23 is entered, .$ returns -$123.23.

Notice also, that the position of SIGN the within definition was changed to place the minus sign at the very beginning of the pictured string.

THE FORTH ASSEMBLER

Now we're going to deal with a slightly more advanced * subject: The FORTH Assembler, and FORTH assembly language programming. FORTH assembly code is a lot like standard 6502 assembly code, except that FORTH assembly code maintains the "Structured" programming approach of FORTH. What this means is you will not see any of the standard 6502 branching commands in FORTH Assembly language (BEQ, BNE, BMI,...). Instead, FORTH Assembly language uses the same IF/THEN, BEGIN/UNTIL, BEGIN/AGAIN structure as FORTH. More on this in a minute. First, lets see how to define a machine language word using the FORTH Assembler.

The first thing to remember is that FORTH Assembly language definitions start with the word CODE. This tells the FORTH Assembler that what follows is a FORTH Assembly language definition. After the word CODE comes the name of the definition, then the machine language instructions, and finally, the command END-CODE to end the FORTH Assembly language definition.

EXAMPLE :

```
CODE TEST    ( THE NAME OF OUR FORTH ASSEMBLY LANGUAGE WORD IS TEST)
45 # LDA,    ( LOAD THE ACCUMULATOR WITH 45 )
32768 STA,   ( STORE THE ACCUMULATOR AT MEMORY LOCATION 32768 )
NEXT JMP,    ( RETURN FROM ASSEMBLY LANGUAGE TO FORTH )
END-CODE     ( END DEFINITION )
```

The neat thing about the FORTH Assembler is the really beautiful way that the machine code blends with SUPER FORTH. Since our little machine language definition was defined using the FORTH Assembler, all you need do to get it to execute is to type its name as you would with any FORTH word:

EXAMPLE :

```
TEST <CR>   ( EXECUTE MACHINE LANGUAGE ROUTINE)
```

To see that "TEST" has worked, type:

```
32768 @ . <CR>
```

This will print the contents of memory location 32768, which should now be 45.

Remember, FORTH Assembly words can be freely used in and

*** This section is for those already familiar with assembler code.**

with standard FORTH words.

There are a couple of things to note here in our example:

1. Notice that CODE and END-CODE act much the same as the colon and semi-colon in standard FORTH.

2. Notice that the commands are entered in Reverse Notation: Instead of LDA # 45, you enter 45 # LDA,. The rule is: Operand first, followed by operator.

3. Notice that each operator is followed by a comma: LDA, STA, JMP,.

4. A FORTH assembly language CODE definition MUST have an instruction at the end of the routine that will send it back to the FORTH operating system. The NEXT JMP, command is one of these EXIT ROUTINES. Others are: POP JMP,, POPTWO JMP,, PUSH JMP, and PUT JMP, (Sounds like Olympic track and field events!). These different EXIT ROUTINES allow you to pass information between your FORTH Assembly language words and the FORTH operating system. You will find them covered in detail in the main manual, in the FORTH Assembler section.

BRANCHING

As stated earlier, FORTH Assembly language does not use the branching commands used in 6502 Assembly language. Instead, it uses the structured looping words of FORTH in conjunction with the 6502 status register.

EXAMPLE : We´re going to use one of the COMMODORE KERNAL routines to print two hundred and fifty six A´s on the screen...

```
HEX            ( SET HEXIDECIMAL NUMBER BASE.)
CODE AAA       ( AAA IS THE NAME OF THE ROUTINE.)
XSAVE STX,     ( SAVE X REGISTER.)
0 # LDX,       ( LOAD X REG. WITH 0.)
41 # LDA,      ( LOAD ACCUM. WITH ASCII VALUE FOR "A" IN HEX)
BEGIN,         ( BEGIN THE LOOP)
FFD2 JSR,      ( KERNAL ROUTINE TO PRINT CONTENTS OF ACCUM.)
INX,           ( INCREMENT X REG..)
0 # CPX,       ( COMPARE X REG. TO 0.)
0= UNTIL,      ( EXIT LOOP ONLY IF X REG.=0.)
```

```
XSAVE LDX,     ( RESTORE X REGISTER.)
NEXT JMP,      ( EXIT ROUTINE.)
END-CODE       ( END DEFINITION.)
```

After entering the above definition, type: AAA <CR>, and part of the screen will be covered with A´s.

The 0= in the third to the last line is a command that tests the ZERO flag of the 6502 STATUS REGISTER. This along with the BEGIN-UNTIL is equivalent to the BEQ command in 6502 Assembly Language. There are other flag words: 0<, 0< NOT, CS, CS NOT, VS and VS NOT. These words are covered in the main manual.

Note that in the above example, we saved the X register. This is one of the few ENTRY/EXIT conditions you have to be careful of. The X register is used to keep track of the FORTH parameter stack. Therefore, whenever you write a machine language definition that might effect the X register, save the contents of the X register by storing it in the special memory location XSAVE, and then restore it just before the EXIT ROUTINE by loading the X register from XSAVE.

Also note that just like all other FORTH Assembly Commands, "BEGIN," and "UNTIL," both are followed by commas. The same is true for all FORTH Assembly Language branching instructions.

One final note: FORTH Assembly Language definitions may be put on screens, and you can put more than one FORTH Assembly Language command on a line. Also, you can use variable names, instead of numerical addresses in FORTH Assembly Language words. Some of the standard FORTH words can be used within your FORTH Assembly Language words, and you can even create and use MACROs. Again, all of this is covered in the main manual.

USING YOUR OWN MACHINE LANGUAGE ROUTINES WITH SUPER FORTH

It´s possible to use machine language routines with SUPER FORTH that were not created with the SUPER FORTH Machine Language Assembler. The following is a description of the procedure. It is assumed that you have the routine on disk in a machine language program file.

1. Make sure that your machine language routine ends with an RTS.

2. Create an area in memory large enough for your machine language routine, using the FORTH words "CREATE" and "ALLOT".

3. Use LOADRAM to load your machine language routine into the newly created area.

4. Call the routine by using GO.

EXAMPLE :

We will load from disk, a hypothetical machine language routine called "TEST1" which is exactly one hundred bytes long.

We have already made sure that TEST1 ends in an RTS.

```
CREATE  HOLE 100 ALLOT <CR> ( CREATE AREA IN DICTIONARY.)

HOLE " TEST1" LOADRAM  <CR> ( TEST1 IS NOW IN HOLE)
```

Now type:

```
HOLE GO <CR>
```

and the routine will execute.

We can even make this a FORTH word, by including the address and the GO instruction into a definition:

```
: TEST1  HOLE GO ;
```

Now, whenever the FORTH word TEST1 is executed, the machine language routine at HOLE will execute. SLICK!

A FEW FORTH PROGRAMMING TECHNIQUES.

The following is a list of programming techniques commonly used in FORTH programming:

1. Whenever possible, try to do it on the stack. Try to stay away from using large numbers of variables as pointers, flags and indexes.

2. Make your definitions as short as possible. If a single colon definition is starting to fill up a screen, it´s better FORTH strategy to break it up into several words.

3. It´s better to place fewer commands on a line and use more lines and more screens, then to pack FORTH screen lines completely full.

4. Whenever possible, use comments. You´ll be glad you did later. Remember, unlike BASIC REM statements, FORTH comments aren´t compiled into definitions, and therefore they do not slow down execution.

5. Indent nested LOOPS and nested IF/THEN statements. This makes it a lot easier to see what´s going on.

EXAMPLE :

```
SCREEN #1
0)
1) : TEST  100 0 DO ." THIS IS" CR
2)              10 0 DO ." SUPER" CR LOOP ( WE INDENT INNER LOOP)
3)      ." FORTH" CR LOOP ;
4)
5)
```

6. To take best advantage of the TOP DOWN programming structure of FORTH, it´s better to start programming on paper, rather than on the screen as you might with BASIC.

7. On FORTH screens, line zero is usually reserved for a comment, and not program code. This allows you to use the word INDEX to search through a group of FORTH screens quickly and easily.

79-STANDARD REFERENCE WORD LIST

The following is a short list of a few of the more commonly used 79-STANDARD FORTH words. This is by no means a complete list. For a complete list, consult the main manual.

ABS	Change the top value on the stack into its absolute value.
ALLOT	Allocate a number of bytes in the dictionary for a list, or table.
AND	Logical AND.
BEGIN	Starts a loop of an unpredictable number of times.
BLOCK	Brings data stored on disk into memory.
C!	Store a byte into the address on the top of the stack.
C@	Fetch a byte from the address on the top of the stack.
CONSTANT	Defines a name for a value which doesn´t change.
DECIMAL	Sets to ten the base of displayed numbers.
DEPTH	Calculate how many values are on the stack.
DO	Start a loop of a known number of times.
DROP	Discards the top value on the stack.
DUP	Duplicates the top value on the stack.
ELSE	Marks a program path to execute if decision value is false.
EMIT	Send a character to the display.
EMPTY-BUFFERS	Erase the storage area reserved for disk blocks.
EXPECT	Obtain a number of characters form the keyboard.

FLUSH	Write data in memory out to disk.
FORGET	Remove user defined word from dictionary.
HEX	Sets to 16 the base for displayed numbers.
IF	Marks the start of a branch or decision choice.
KEY	Obtain one character from keyboard.
LEAVE	Marks a midpoint exit for a DO-loop.
LIST	Display the the text contents of a storage block (Displays a FORTH Editing Screen).
LOAD	Compiles into the dictionary a program stored in text form on disk.
LOOP	Marks the end of a DO-loop.
MOD	Divide two values on the stack, leaving only the remainder.
NOT	Reverses a truth value from true to false.
OR	Logical OR.
OVER	Duplicates the second stack value and places it on the top of the stack.
REPEAT	An alternate ending for a BEGIN-loop. Used with WHILE.
ROT	Rotate the third number on the stack to the top.
SPACE	Print a space.
SWAP	Interchanges the top two numbers on the stack.
THEN	Marks the end of an IF...ELSE...THEN decision.
UNTIL	Marks the end of a BEGIN-loop.
VARIABLE	Defines a name for a two byte storage cell.
WHILE	Marks a mid-loop exit in a BEGIN/REPEAT loop.
>R	Moves a number from the parameter stack to the return stack.

R>	Moves a number from the return stack to the parameter stack.
:	Create a dictionary entry for the word following the colon.
;	Ends the dictionary entry for the word.
*	Multiplies two numbers on top of the stack.
/	Divides two numbers on top of the stack,
/MOD	Divides two numbers, leaving both quotient and remainder.
+	Adds two numbers on top of the stack.
-	Subtracts two numbers on top of the stack.
!	Stores a two byte value into an address.
+!	Adds the number on the top of the stack to the contents of an address.
@	Fetches a value from an address.
'	Returns the Parameter Field Address of a word.
=	Compares two values on the stack for equality.
0=	Test if top value on stack is equal to zero.
<	Test if second stack value is less than top stack value.
0<	Test if top stack value is less than zero,
>	Test if second stack value is greater than top stack value.
0>	Test if top stack value is greater than zero.
1+	Increases top stack value by one.
1-	Decreases top stack value by one.
2+	Increases top stack value by two.

2-	Decreases top stack value by two.

BASIC TO SUPER FORTH TRANSLATION TABLE

The following table is intended only as a stepping-stone to get BASIC programmers thinking in terms of FORTH. Do not make the mistake of using BASIC programming strategy for FORTH programming. FORTH structured programming is more efficient and more powerful (See section on structured TOP-DOWN/BOTTOM-UP PROGRAMMING).

BASIC	SUPER FORTH
ABS :	ABS :
Return absolute value of number.	Return absolute value of a number on the stack.
AND :	AND :
Logically AND two numbers	Logically AND two numbers on the stack.
ASC :	NO SIMILAR COMMAND *
Return ASCII value of a character in a string.	
ATN :	FATN :
Returns arctangent of a number.	In floating point mode, returns arctangent of a number on the stack.
CHR$:	NO SIMILAR COMMAND * EMIT
Convert ASCII code to character.	
CLOSE :	CLOSE :
Close an opened channel.	Close an open channel.
CLR :	NO SIMILAR COMMAND *
Re-allocate memory.	

CMD :	CMD CMDI :
Re-direct output.	Re-direct input and output.
CONT :	NO SIMILAR COMMAND *
Re-start program.	
COS :	FCOS :
Returns cosine of a number.	In floating point mode, returns cosine of a number on the stack.
DATA :	C, , :
Store information within a program.	Store information within a dictionary location.
DEF FN :	NO SIMILAR COMMAND *
Define a function.	
DIM :	1ARRAY 2ARRAY MDIM LDIM :
Define an array.	Define various dimensional matrices.
END :	EXIT QUIT :
End a program.	Exit execution of FORTH words.
EXP :	FEXP :
Return e to the power of x.	In floating point mode, return e to the power of the number on the stack.
FRE :	FRE :
Return amount of available RAM.	Return amount of available RAM.
GET :	?TERMINAL :
Read keyboard.	Read keyboard.
GET# :	GET# :

Get character from a file.	Get character from a file.
GOSUB :	NO SIMILAR COMMAND *
Go to subroutine.	
GOTO :	NO SIMILAR COMMAND *
execute program out of sequence.	
IF...THEN... :	IF...ELSE...THEN... CASE :
Perform a decision.	Perform a decision.
INPUT :	INPUT KEY $INPUT :
Input information.	Input various forms of information.
INPUT# :	INPUT# :
Input information from a file.	Input information from a file.
INT :	FINT :
Return the integer value of a number.	In floating point mode, return the integer value of a number on the stack.
LEFT$:	LEFT$:
Return left string.	Return left string.
LEN :	$LEN :
Return length of string.	Return length of string.
LIST :	LIST DECOMPILE :
List program.	List FORTH screens, decompile FORTH word.
LOAD :	LOAD :
Load a program into memory.	Load a set of FORTH screens and compile them into memory.

LOG :	FLOG FLOGX :
Return natural log of a number.	In floating point mode, return various log functions of a number on the stack.
MID$:	MID$:
Return middle string.	Return middle string.
NEW :	FORGET EMPTY :
Delete program.	Delete FORTH words.
NEXT :	DO...LOOP BEGIN...AGAIN BEGIN...UNTIL BEGIN...WHILE... :
Looping structure.	Various looping structures.
NOT :	NOT :
Reverse logical value.	Reverse logical value.
OPEN :	OPEN :
Open a file.	Open a file.
OR :	OR :
Logically ORs two numbers.	Logically ORs two numbers.
PEEK :	@ :
Reads a memory location.	Fetch value from memory location, placing it on the stack.
POKE :	! :
Store a value in memory.	Store a value in memory.
PRINT :	. $. .S EMIT :
Print a string or value.	Various printing commands.
PRINT# :	PRINT# :
Print to a file.	Print to a file.

READ :	NO SIMILAR COMMAND *
Read DATA statements.	
REM :	( ) :
Commentary.	Commentary.
RESTORE :	NO SIMILAR COMMAND *
Reset DATA pointer.	
RETURN :	LEAVE :
Return from subroutine.	Exit a DO loop.
RIGHT$:	RIGHT$:
Return right string.	Return right string.
RUN :	"Type name of word" :
Run program.	Execute FORTH word.
SAVE :	FLUSH SAVE-BUFFERS SAVE-FORTH APPLICATION :
Save a program.	Various saving commands.
SGN :	FSGN :
Returns signum value of a number.	In floating point, mode returns signum value of number on the stack.
SIN :	FSIN :
Returns sine value of a number.	In floating point mode, returns sine of a number on the stack.
SPC :	SPACES :
Print spaces.	Print spaces.
SQR :	FSQR :
Return square root of a number.	In floating point, return square root of a number.

STATUS :	ST :
Return I/O status.	Return I/O status.
STR$:	<# #S SIGN HOLD #> :
Convert numeral to ASCII string.	Converts numerals to ASCII string.
SYS :	SYS SYSCALL :
Jump to memory location.	Jump to memory location.
TAN :	FTAN :
Return the tangent of a number.	In floating point mode, return the tangent of a number on the stack.
VAL :	$VAL :
Return numeric value of a string.	Return numeric value of a string.
WAIT :	BEGIN...UNTIL BEGIN...WHILE :
Wait for condition.	Loop until condition.

*=These commands are either unnecessary, or FORTH performs them differently.

Index

#BUFF 15
--> 15
. 8
79-STANDARD WORD LIST 30
: 11
; 11
APPLICATION 17
backup copy 7
BASIC translations 34
characters per line 13
CODE 25
colon definition 11
comments 14
DOT 8
EMPTY-BUFFERS 15
END-CODE 25
exit routines 26
extending the system 17
F 15
FIRST-IN-LAST-OUT structure 8
FORTH Assembler 25
FORTH Assembler looping words 25
FORTH Assembler syntax 26
FORTH Assembly Branching 26
FORTH EDITOR SCREENS 13
FORTH Extensions 13
FORTH math operation 9
FORTH versus BASIC 8
FORTH word syntax 9
GO 28
lines per screen 13
LIST 14
LOAD 15
loading machine language files 27
loading other programs 27
loading screens 15
LOADRAM 28
master disk 7
OK 11
programming technique 29
relative files 17
SAVE-FORTH 7

saving FORTH screens 15
screen zero 18
SQUARE 19
stack 8
stack manipulation words 9
stack order 8
STARTING FORTH 2
status register 27
structured programming 25
THRU 15
TOP-DOWN/BOTTOM-UP PROGRAMMING 19
W 14
WIPE 14
word list 30
working copy 7
X register 27
XSAVE 27