# FHL eforth

FHL FRANK HOGG LABORATORY
THE REGENCY TOWER • SUITE 215 • 770 JAMES ST. • SYRACUSE, NY 13203
PHONE (315)474-7856 • TELEX 646740

# A "Tour De FORTH"

with

eFORTH

by Charles E. Baker

copyright 1983

Frank Hogg Laboratory

## MANUAL REVISION HISTORY

| Revision | Date | Change |
|---|---|---|
| A | 25oct83 | Original Release, eFORTH 1.0 |

## COPYRIGHT INFORMATION

This entire manual is provided for the personal use and enjoyment of the purchaser. Its contents are copyrighted by Charles E. Eaker and Frank Hogg Laboratory, Inc., and reproduction in whole or in part, by any means, is prohibited. Use of this program, or any part thereof, for any purpose other than single use by the purchaser is prohibited.

## DISCLAIMER

The supplied software is intended for use only as described in this manual. Use of undocumented features or parameters may cause unpredictable results for which neither Charles E. Eaker nor Frank Hogg Laboratory, Inc. can assume responsibility. Although every effort has been made to make the supplied software and its documentation as accurate and as functional as possible, Charles E. Eaker and Frank Hogg Laboratory, Inc. will not assume responsibility for any damages incurred or generated by such material. Charles E. Eaker and Frank Hogg Laboratory, Inc. reserve the right to make changes in such material at any time without notice.

## INSTALLING eFORTH

To get eFORTH up and running on your computer, follow the instructions in the Appendix which applies to your operating system or computer.

copyright 1983 

# TABLE OF CONTENTS


| | | | |
|---|---|---|---|
| 1 | **WHY FORTH?** | | **8** |
| | THE FORTH ENVIRONMENT | 8 | |
| | THE FORTH PHILOSOPHY | 8 | |
| | THE FORTH COMMUNITY | 9 | |
| 2 | **HOW DO YOU SAY "HELLO"?** | | **10** |
| | THE FORTH INTERPRETER | 10 | |
| | TYPING MISTAKES | 11 | |
| | WORDS | 11 | |
| | STOPPING THE OUTPUT | 12 | |
| | THE DICTIONRY AND ITS VOCABULARIES | 12 | |
| | CONTEXT AND CURRENT | 12 | |
| | MORE WORDS | 13 | |
| | REDEFINING A WORD | 14 | |
| | FORGETTING A WORD | 15 | |
| | EVEN MORE WORDS | 15 | |
| | DEFINE BEFORE USE | 15 | |
| | STARTING FORTH WITH eFORTH | 16 | |
| 3 | **WHAT DO YOU SAY AFTER YOU'VE SAID "HELLO"?** | | **18** |
| | NUMBERS | 18 | |
| | EMPTY STACK | 20 | |
| | CONSTANTS | 21 | |
| | VARIABLES | 21 | |
| | AN AVERAGE EXAMPLE | 22 | |
| | MANIPULATING THE STACK | 23 | |
| | DECIMAL - BASE TEN | 24 | |
| | HEXADECIMAL - BASE SIXTEEN | 24 | |
| | BINARY - BASE TWO | 25 | |
| | CHOOSING NAMES | 25 | |
| 4 | **WHAT CAN I DO WITH IT?** | | **28** |
| | GLOSSARY ENTRIES | 29 | |
| | LOOK, MA! NO VARIABLES | 29 | |
| | THE RETURN STACK | 30 | |
| | FOOD FOR THOUGHT | 31 | |
| | DEFINING A WORD THAT DEFINES OTHER WORDS | 32 | |
| | WHAT DOES does> DO? | 32 | |
| | GETTING FANCIER OUTPUT | 33 | |
| | USING FANCIER INPUT | 34 | |
| | DOUBLE NUMBERS | 34 | |
| | IT'S THE PHONE AGAIN | 36 | |


 copyright 1983

| | | | |
|---|---|---|---|
| 5 | **HOW DO I SAVE AND EDIT MY DEFINITIONS?** | | 38 |
| | THE FORTH MEETS THE DISK | 38 | |
| | PUTTING TEXT ON A BLOCK | 39 | |
| | THE CURRENT BLOCK | 40 | |
| | THE CURRENT LINE | 40 | |
| | REPLACING AND DELETING LINES | 41 | |
| | THE INSERT BUFFER | 42 | |
| | STRING EDITING COMMANDS | 42 | |
| | THE FIND BUFFER | 42 | |
| | HOW TO INTERPRET A BLOCK | 43 | |
| | ERRORS WHILE LOADING | 44 | |
| | ANSWERING THE PHONE PROBLEM | 44 | |
| | BACK TO THE RESTAURANT | 45 | |
| | HOW DID YOU DO? | 45 | |
| | THE ANSWERS, PLEASE | 46 | |
| | ELIMINATING CRAMPS | 46 | |
| | BLOCK EDITING COMMANDS | 47 | |
| | DOCUMENTING YOUR APPLICATION | 47 | |
| 6 | **DOES FORTH HAVE WHAT COUNTS?** | | 50 |
| | LET ME COUNT THE A's | 50 | |
| | HOW DO LOOPS WORK? | 51 | |
| | DO THE I's HAVE IT? | 52 | |
| | CAN I MAKE IT RUN FASTER? | 53 | |
| | DON'T GO OUT OF BOUNDS | 53 | |
| | WHAT'S YOUR SINE? | 54 | |
| | IF...THEN | 55 | |
| | IF...ELSE...THEN | 56 | |
| | WHAT DOES YOUR SINE LOOK LIKE? | 56 | |
| | INDEFINITE LOOPS | 57 | |
| | SOME ODDS AND ENDS | 59 | |
| | IT'S TIME TO leave | 59 | |
| 7 | **WHAT'S IN A WORD?** | | 62 |
| | THE LINK FIELD | 62 | |
| | THE NAME FIELD | 63 | |
| | THE CODE FIELD | 63 | |
| | THE PARAMETER FIELD | 63 | |
| | VARIABLES | 63 | |
| | CONSTANTS | 64 | |
| | COLON DEFINITIONS | 65 | |
| | COMPILATION | 66 | |
| | IMMEDIATE WORDS | 67 | |
| | COMPILE TIME AND RUN TIME | 68 | |
| | COMPILE TIME | 68 | |
| | RUN TIME | 68 | |
| | CODE DEFINITIONS | 69 | |

| | | | |
|---|---|---|---|
| 8 | **HOW DOES FORTH WORK?** | | **70** |
| | THE FORTH MACHINE'S REGISTERS | 70 | |
| | WHO'S NEXT? | 71 | |
| | IMPLEMENTING THE FORTH MACHINE | 71 | |
| | THE eFORTH 6809 FORTH MACHINE | 72 | |
| | THE INTERPRETER | 72 | |
| 9 | **HOW DOES FORTH COMPILE NUMBERS?** | | **76** |
| | NUMERIC LITERALS | 76 | |
| | BRANCHING | 77 | |
| | WHEN if COMPILES | 80 | |
| | HOW compile WORKS | 80 | |
| | STRING LITERALS | 82 | |
| 10 | **VOCABULARIES** | | **84** |
| | CONTEXT AND CURRENT VOCABULARIES | 84 | |
| | CREATING NEW VOCABULARIES | 84 | |
| | VOCABULARY CHAINING | 84 | |
| | DICTIONARY SEARCHING | 85 | |
| | SEALED VOCABULARIES | 86 | |
| 11 | **HOW CAN I PROTECT MYSELF?** | | **88** |
| | COMPILER SECURITY | 88 | |
| | DISK ERRORS | 89 | |
| | EXECUTION VARIABLES | 89 | |
| 12 | **THE eFORTH 6809 ASSEMBLER VOCABULARY** | | **90** |
| | code DEFINITIONS | 90 | |
| | ;code DEFINITIONS | 92 | |
| | BRANCH INSTRUCTIONS AND PROGRAM STRUCTURE | 93 | |
| | eFORTH ASSEMBLER SYNTAX | 94 | |
| | IMMEDIATE ADDRESSING | 95 | |
| | EXTENDED ADDRESSING | 95 | |
| | DIRECT ADDRESSING | 95 | |
| | INDEXED ADDRESSING | 95 | |
| | RELATIVE ADDRESSING | 97 | |
| | 6889 MNEMONICS | 97 | |
| | MNEMONICS - NO OPERANDS | 97 | |
| | MNEMONICS - IMMEDIATE ADDRESSING ILLEGAL | 98 | |
| | MNEMONICS - IMMEDIATE ADDRESSING PERMITTED | 98 | |
| | MNEMONICS - IMMEDIATE OPERANDS REQUIRED | 98 | |
| | MNEMONICS - INDEXED ADDRESSING REQUIRED | 98 | |
| | MNEMONICS - REGISTER OPERANDS REQUIRED | 99 | |
| | MACROS | 99 | |


 copyright 1983

**13 WHERE DOES eFORTH PUT THINGS? 100**
THE DICTIONARY 100
THE PARAMETER STACK 101
THE TERMINAL INPUT BUFFER 101
THE RETURN STACK 101
THE DISK BUFFERS 101
THE USER VARIABLE AREA 101

**14 THE END OF THE TOUR 102**
LITERAL STRINGS 102
SMART WORDS 103
A CASE STRUCTURE 103

**APPENDICES**

A HOW DOES eFORTH DIFFER FROM "Starting FORTH"?
B THE eFORTH MASTER GLOSSARY
C LISTINGS - eFORTH STANDARD EXTENSIONS AND ELECTIVES
D eFORTH INSTALLATION - FLEX
E eFORTH INSTALLATION - TRS80 COLOR COMPUTER

# CHAPTER 1

# WHY FORTH?

Why would anyone choose to use FORTH to write programs instead of a better-known language such as FORTRAN or Pascal or even BASIC which probably came free with the computer? FORTH is more than a programming language. It is a programming environment, and it is a programming philosophy.

## THE FORTH ENVIRONMENT

FORTH is a "modeless" environment. At any given moment, the FORTH disk operating system and its commands are available to you. So are the FORTH editing commands, the FORTH compiler, the FORTH interpreter, and the FORTH assembler. These are not separate programs that you have to "get out of" in order to use one of the others. The resources of each are available to the others at all times.

FORTH is extensible. This means that you can build new commands, new functions, and new data structures out of existing ones. The new ones look and behave like the old ones.

FORTH is interactive. You can create and immediately test new commands, functions, and data structures from the keyboard. In FORTH, your "programs" are written in small pieces called "words" that are combined to make new ones. Any word can be tested from the keyboard. If what you are testing needs data, you can supply it from the keyboard. If it returns data, you can see what comes back at the keyboard.

## THE FORTH PHILOSOPHY

The FORTH philosophy is based on this principle:

**Only you should protect yourself from your mistakes.**

Unlike other languages, FORTH does not stop running your program and tell you that you tried to do something that is "wrong" and, in its infinite wisdom, has prevented a "terrible" thing from happening. On the contrary, FORTH will let you divide by zero,

overflow arithmetic operations all over the place, and all sorts of other "evil" things.

Neither Pascal nor BASIC, for example, will allow you to directly get the sum of an integer with a character. That's a "type" error. Presumably, it's an operation that doesn't make sense. But both languages give you roundabout ways of doing it because it is often a valuable thing to do. FORTH doesn't care. FORTH holds **you** responsible for the correctness of your programs. FORTH does not assume that it knows better than you what a programming error is.

If you are a bad programmer in other languages, you may well be a terrible FORTH programmer. On the other hand, if you are a bad programmer in other languages, it may be because those languages are forever getting in your way, and much of your time is spent circumventing the language's attempts to "protect" you from yourself.

FORTH never gets in the way because of something built into it. If FORTH is a bother at all, it's because things are missing. Your job is to add them. And while you're at it, you can add things to protect yourself. You, after all, know what kinds of mistakes you tend to make, and you should decide whether to have the computer spend time and effort looking for them.

## THE FORTH COMMUNITY

The community of FORTH users is small but intense, talented, and growing. You can keep up to date with the goings on by joining the FORTH Interest Group. The main membership benefit is **FORTH Dimensions** which is published six times a year. A membership (which includes a subscription) is currently $15 per year. There may even be a FIG chapter in your area. Here's the address.

**FORTH Interest Group**
**P.O. Box 1105**
**San Carlos, CA 94070**

# CHAPTER 2

## HOW DO YOU SAY "HELLO"?

One of the first things people often have a computer do is simply say "hello". You have probably been attacked by this primordial urge already, but you don't know how to do it in FORTH. Here's how. Enter

```
: hi  ." Hello, Dummy!"  ;
```

and hit the "return" key or "enter" key or whatever key your computer has for you to push when you finish typing a line of input. Be sure you include the spaces, and be sure you include the semicolon at the end. Unlike BASIC, spaces are crucial in FORTH.

When you hit the return key, FORTH responds by saying "ok". Now enter **hi** and FORTH will print "Hello, Dummy!" and that's all there is to it.

You have just written your first FORTH program. Actually, FORTH programmers don't "write programs"; they "define words". So, you have just defined your first word in FORTH, and it's name is **hi** .

The definition of a word obviously begins with a colon followed by the name of the new word. Then we include the names of the words to be executed when the new word is executed, and the definition is terminated with a semicolon.

This means that ." (pronounced "dot-quote") is a FORTH word. It can only be used in a definition. What it does is to arrange things so that the string which follows it will be printed when **hi** is executed. In fact, it's the only FORTH word which is used in the definition of **hi** .

### THE FORTH INTERPRETER

You are communicating with the FORTH interpreter. After you type a line of FORTH words, the interpreter executes them, one after the other, from left to right, then says "ok". However, the interpreter can only execute the words in the input if it can find them in its "dictionary". If you type in a word it can't

find, it will complain.

### TYPING MISTAKES

Did you make a typing mistake and get an error message instead of an "ok"? No problem. Just enter the whole line again, but there may be an easier solution.

First, make sure that your keyboard is generating both lower and upper case letters. To eFORTH, "hi" and "HI" and "hI" and "Hi" are all different.

Did you mis-type just one character in the middle of the line? Hold down the "control" key, then press the "A" key, then release the "A" key and the control key. (In the future, we will simply refer to this sequence as "control-A".) The last line you entered is printed out again. Use the backspace key to get back to the character you messed up. Replace it with the correct character. Now, hit "control-A" again, and you will see the tail end of the line you backspaced over. Now you can hit the return key just as if you had re-typed the entire line.

Did you notice a typing error before you hit the return key? Use the backspace key to move the cursor back to the mistake and re-type the line from that point.

Is the mistake so bad that you'd just as soon scratch the whole line and start over? Hold down the "control" key, then press the "X" key, then let up on the "X" key and the "control" key ("control-X"). The line will disappear. Now try typing it again.

### WORDS

FORTH is just a collection of words, and any word in FORTH can be executed or it can be used in the definition of a new word. Do you want to see some of the FORTH words which have been defined for you? Enter **forth words** and hit your "return" or "enter" key.

Look at all those words! What do they all do? You will find out soon enough, but it may turn out that none of them do anything you want your computer to do for you. If so, just add your own words to the list by defining them to do whatever you want done.

Look, there's **hi** in the list. It's the first one. If you look real hard, you will also find **."** somewhere in that mess.

## STOPPING THE OUTPUT

Did the list of words fly by too fast for you? You can stop the output by hitting the "escape" key on most terminals. Use the "break" key on the Color Computer. When you are ready for more output, press the "escape" key again. You can terminate the output operation all together by hitting the "return" key.

## THE DICTIONARY AND ITS VOCABULARIES

The portion of memory where all of the words are stored is called "the dictionary", and each word in the dictionary is assigned to a "vocabulary". The words we just listed are all in the **forth** vocabulary. In eFORTH there are four others: **system editor assembler** and **disking** . You can see the words in each of those vocabularies by entering the name of the vocabulary followed by **words** .

How large is the dictionary? Enter

```
here origin - u.
```

and hit return. Don't forget the dot, and don't forget the spaces. You will see the number of bytes of memory presently consumed by all the words in the dictionary. Each time we add a word to the dictionary this number increases.

## CONTEXT AND CURRENT

Whenever we enter the name of a vocabulary, that vocabulary becomes the "context" vocabulary which means that the interpreter will always search that vocabulary first. If the context vocabulary is not also the **forth** vocabulary, then the **forth** vocbulary is searched next (and last). In other words, the **forth** vocabulary is always searched. The details of this are discussed in a later chapter.

The "current" vocabulary is the vocabulary to which new words are added. Let's add a word to the **system** vocabulary. Enter the following line

```
system definitions
```

and hit return. The interpreter first executes **system** which makes the **system** vocabulary the context vocabulary, and then the interpreter executes **definitions** which sets the current vocabulary to be whatever the context vocabulary happens to be at the time. Now enter

**: status? cr ." Buzz off, Turkey!" ;**

and you can amuse yourself by asking a friend to check the status of your computer by typing in **system status?** and hitting the return key. The **cr** simply starts printing on a new line.

Before going on, enter **forth definitions** and hit return. Now enter **status?** and note that it's not there. The interpreter can't find it unless we first make the **system** vocabulary the context vocabulary.

**MORE WORDS**

Want to print a single character? Enter **cr 65 emit** and hit the return key. FORTH will start a new line and print an "A" on it. A whole pile of "A's" can be printed with a loop. Enter

**: chars cr 0 ?do 65 emit loop ;**

Now enter **10 chars** and hit the return key. Let's make **chars** a little fancier. But first, let's get rid of the old one. Enter **forget chars** and hit the return key. Now enter

**variable char 65 char !**

and hit the return key, then enter

**: chars cr 0 ?do char @ emit loop ;**

and hit the return key. Once again enter **10 chars** and hit the return key. The result is the same, right? Now enter **45 char !** and hit the return key. Don't forget the exclamation mark. Now enter **10 chars** and see what you get. Change the 10 to some other number. Change the value of **char** to some other character.

We have been using decimal numbers for the ASCII characters. Perhaps you are more accustomed to expressing them with hexadecimal numbers. Enter **hex** and hit the return key. Now enter **40 char !** and hit the return key. 40 is the ASCII hexadecimal code for the "at" sign. What do you suppose entering **10 chars** and hitting enter will print out? Sixteen of them because the hexadecimal number 10 equals sixteen. Would you rather have the numbers you enter be interpreted as decimal

numbers again? Enter **decimal** and hit the return key.

Would you like to set the value of **char** with hexadecimal numbers and call **chars** with decimal numbers? Ok, the ASCII hex code for an up arrow is 5E, right? And you want to print out 20 (decimal) of them? Enter

```
hex 5E char ! decimal 20 chars
```

and hit the return key.

Have you ever worked with a language as congenial as FORTH; one that does what you tell it to do and says "ok" every time? It's interactive just like BASIC, and it lets you use names that are far more descriptive than "F2" or "A$". In fact, if you don't like the name of a word in FORTH, change it. You can rename **chars** to be something like **characters** or whatever name you prefer, very easily by entering

```
: characters chars ;
```

and hitting the return key. Now whenever you enter **characters** it does the same thing as **chars** . Prefer a shorter name to save your fingers? enter

```
: c chars ;
```

hit the return key, and entering **10 c** will do the same thing as **10 chars**

You don't even have to restrict your names to numbers and letters. Enter

```
: $ chars ;
```

and it will be "ok" with FORTH. Now **10 $** will do the same thing as **10 chars** . The names of your words can be any combination of characters in any order you like.

## REDEFINING A WORD

You can define another word named **chars** if you wish. Enter

```
: chars cr cr chars cr cr ;
```

Then enter **chars** and see what it does.

The new **chars** skips two lines, then executes the old **chars** , then skips two more lines. Enter **words** again, and you will see that **chars** appears in the list twice. However, only the last one you defined can now be executed or used in a definition.

## FORGETTING A WORD

If you enter **forget chars** , the last one you defined will be removed from the dictionary. Try it. Then enter **words** to verify that it is gone. If you enter **forget chars** again, the first one you defined will be removed. However, **forget** will also remove every word you have defined since you defined **chars** . You cannot selectively remove words from the dictionary; only a word and all words defined since it was defined.

## EVEN MORE WORDS

Want to crash once in a while? Define a word to do it.

```
: crash begin again ;
```

Now, whenever you enter **crash** the only escape is to hit the reset button. A bit drastic, perhaps, but it makes the point that everything is easy in FORTH.

A less dramatic capability is ordering your computer to sleep. Enter

```
: sleep begin snore key? until ;
```

and you will get an error message because **snore** has not been defined.

## DEFINE BEFORE USE

FORTH obeys the rule "Define Before Use" without exception. You cannot execute a word which is not in the dictionary (has not been defined), and you cannot use a word in a definition which is not in the dictionary.

So, let's define **snore**

```
: snore cr ." Z Z Z Z Z" ;
```

Now, enter the definition of **sleep** again, and when your machine

is getting on your nerves just tell it to **sleep** . You can wake it up again by gently nudging one of its keys.

Enough of this foolishness. All of the foregoing nonsense has given you a quick taste of FORTH, but it has not given you much that's useful in learning how to use FORTH to express the demonic schemes lurking in the recesses of your own mind. You will have to learn the "ok" way to use words, numbers, and lots of other things in FORTH. This manual will take you on a quick tour of FORTH.

A more comprehensive introduction to FORTH is **Starting FORTH** by Leo Brodie and published by Prentice-Hall. It can be ordered from Frank Hogg Laboratory. **Starting FORTH** will also show you some interesting things about the internal workings of computers. It is an excellent introduction to both FORTH and computers.

## STARTING FORTH WITH eFORTH

If you decide to use **Starting FORTH** with eFORTH, there are a few differences between eFORTH and the FORTH which Brodie uses which you should be aware of. Most of them involve subtle and advanced features of FORTH which you don't have to worry about right now. Every word which Brodie uses in his examples and exercises has been defined for you in eFORTH. A complete list of differences is given in an appendix.

# CHAPTER 3

## WHAT DO YOU SAY AFTER YOU'VE SAID "HELLO"?

FORTH uses a stack for all calculations, holding intermediate results, and passing parameters from one word to another. The stack is a last-in, first-out stack which means that you only have access to the last item which was pushed onto the stack. The phrase "top of the stack" is used to refer to the last item pushed to the stack. Putting things on the FORTH stack is like parking cars in a skinny driveway; you can't get the car in the garage out until all the others have been moved.

Efficient use of the stack requires the use of Reverse Polish Notation (RPN) which takes some getting used to. So, let's start.

Enter **1 29** and hit the return key. FORTH will respond with an "ok" and wait on the next line for more input. "Ok, what?" you may be saying. "What did it do?"

Your keystrokes are read and saved until you hit the return key. After you hit the return key, FORTH attempts to interpret your input, one word at a time. A word in FORTH is any sequence of characters separated by spaces. So, FORTH first finds the word **1** in your input. Now, FORTH searches for it in the dictionary. You may have noticed that **1** is in the dictionary. When **1** is found in the dictionary, the interpreter executes it, and **1** does whatever it was defined to do.

### NUMBERS

**1** is defined to push the binary representation of the integer one to the stack. FORTH stores integers in the computer as 16-bit, binary numbers. If that 16-bit binary number is interpreted as a signed number, it can represent integers in the range from -32,768 to +32,767. If the 16-bit binary number is interpreted as an unsigned number, it can represent non-negative integers in the range from 0 to 65,525.

### CONSTANTS

FORTH allows the declaration of constants. For example, enter the following lines

```
50  constant  fifty
40  constant forty
forty  fifty  +  .
```

and figure out what action is taken by a word which has been defined as a constant. Right. It pushes to the stack the number which that constant has been defined to be. Here is a stack diagram.

| WORD | STACK ----> |
|---|---|
| | (The stack is empty.) |
| forty | 40 |
| fifty | 40 50 |
| + | 90 |
| . | (The stack is empty.) |

### VARIABLES

FORTH also allows the declaration of variables. For example, enter these two lines

```
variable  age
age  .
```

and ponder what a variable does. The first line created a variable named **age** , and **words** will now list it as being in the dictionary. The second line caused **age** to be executed. It put a number on the stack which the dot printed out. What is that number? It is the memory address where the value of the variable named **age** is stored. That's all well and good, but how does one get the value stored at the address of the variable onto the stack? Enter

```
age  @  .
```

You will see the value that **age** was initialized to when it was defined. So, @ (pronounced "fetch") is a word defined to remove an address from the stack, then push the 16-bit contents of that address to the top of the stack.

As we all know, **age** is a variable whose value is updated (with emphasis on the "up") on a periodic basis. How does one assign a new value to **age** ? Enter **34 age !** and hit the return key. Now enter **age ?** and hit the return key. "34" will be printed because **?** is in the dictionary and has been defined as

```
: ? @ . ;
```

which means that instead of typing **@ .** you may simply type **?** and the result is the same.

Notice that when **!** (pronounced "store") is used, the data to be stored is on the stack under the address at which it is to be stored. Here is a stack diagram of an example.

| WORD | STACK ——> |
|---|---|
| | (The stack is empty.) |
| forty | 40 |
| age | 40 address-of-AGE |
| ! | (The stack is empty.) |

Furthermore, **@** and **!** need not be used with the names of variables. If you enter **40 100 !** then 40 will be stored in the two bytes (there are 8 bits in a byte) at addresses 100 and 101. The words **@** and **!** always fetch and store 16-bit numbers. If you wish to manipulate single bytes in memory, the 8-bit memory operations are **c@** and **c!** .

A fancier memory manipulation word is **+!** which is pronounced "plus-store." If you enter **2 age +!** and hit the return key, 2 will be added to the current contents of **age** . Thus, if **age** equals 34 before this operation, it will equal 36 when it is completed. Similarly, **-2 age +!** will subtract 2 from the contents of **age** .

## AN AVERAGE EXAMPLE

Suppose you want the sum and average of several numbers. Suppose the numbers are 280 319 647 12 219 and 572. You can have your results by entering

```
280 319 + 647 + 12 + 219 + 572 + dup . 6 / .
```

and hitting the return key. Actually, you can hit the return key any time you like and as often as you like. If you entered the following four lines,

Back to our line of input. **29** , unlike **1** , is not in the dictionary. Obviously, the interpreter won't find it when it looks for it there. What happens then? The interpreter attempts to interpret the word as a number. **29** can certainly be interpreted as a number. The interpreter then converts it to its internal, binary representation and pushes it to the stack. So, **1** and **29** have both been pushed to the stack.

Let's check it. There is a special word in eFORTH which will print out all of the numbers which are presently on the stack. Enter **.s** and hit return. It should print out

**0 1 29**

followed by the usual "ok".

Now enter **+** and hit return. Again, the interpreter looks for **+** in the dictionary, finds it, and executes it. **+** is defined to remove the top two 16-bit numbers from the stack, add them together (ignoring any overflow), and push their sum to the stack. So, **+** removes **29** from the stack, removes the **1** from the stack, adds them together, and places the result, **30** , on the top of the stack. Use **.s** to verify this.

The interpreter gets the next word, which is the carriage return, and executes it. This results in the printing of "ok," and waiting for a new line of input (which it has been doing while you were reading this).

Now enter a period and hit the return key. The "dot" is a FORTH word which is defined to print the signed, 16-bit number on top of the stack followed by a space. The number is removed from the stack. Ah-ha! FORTH can be used interactively as an RPN calculator. Try some more lines of input.

```
-346  -247  +  .
3  2  *  .
8  5  -  .
10  3  /  .
10  3  mod  .
```

Next, decide what the result will be if you enter

```
4  3  2  +  *  .
```

then try it. Were you right? If not, do you see why? Pretend that you are the FORTH interpreter. The first word in the input stream is 4, so you push 4 to the stack, then 3, then 2. The next word is + so you remove the top two numbers (3 and 2), add them up, and put 5 on top of the stack. The next word is * so you find it in the dictionary and execute it. * is defined to

remove the two top 16-bit numbers from the stack (4 and 5), multiply them, and put the result, 20, on the stack. Finally, the dot is interpreted which prints "20" on the terminal (instead of "14").

Here is a "picture" of what happens when each word is interpreted.

| **WORD** | **STACK ---->** |
|---|---|
| | **(The stack is empty.)** |
| **4** | **4** |
| **2** | **4 2** |
| **3** | **4 2 3** |
| **+** | **4 5** |
| ***** | **20** |
| **.** | **(The stack is empty.)** |

If you have never seen Reverse Polish Notation before, you may find it somewhat odd to express **4 * (3 + 2)** as **4 3 2 + *** and you may think that the equivalent 3 2 + **4 *** is only slightly more "natural". Be assured that if you hang around something long enough, it will soon seem quite "natural". The advantages of RPN are two. First, no parentheses in an expression are necessary, so you can console yourself with the prospect of fewer key-strokes. Secondly, by using a notation which is "natural" for a first-in, last-out stack, we achieve extremely efficient parameter passing from word to word. Stick with it.

**EMPTY STACK**

If you lose track of what is on the stack (and you will from time to time) and you try to print a number from the stack (or remove it in some other way) when the stack is empty, you will get an error message. FORTH is not harmed or bashed by this. Try it. Keep entering dots until you get the "Empty stack." message.

```
280 319 +
647 + 12 + 219
+ 572 + DUP
. 6 / .
```

you would get the same results.

### MANIPULATING THE STACK

This line of input contains a new word. **dup** is defined to push to the stack a copy of the word which is on top of the stack. If you enter **10 dup** there will then be two tens on top of the stack. When **dup** is executed in the above line, the sum we are after is on top of the stack. But we want to print it out, and we also want to use it to calculate the average. So, we copy it, print out the copy, and use the original which is still on the stack to calculate the average. The average is then printed (ignoring the remainder).

Other words that perform operations on the stack are **over** , **drop** and **rot** . Suppose you have

```
10  12
```

on the stack. (We henceforth use the convention of listing the top item on the stack on the right.) If you enter **over** you will have

```
10  12  10
```

on the stack. And if you enter **over** again, you will have

```
10  12  10  12
```

on the stack. If you have

```
1  2  3
```

on the stack, **rot** will give you

```
2  3  1
```

on the stack.

To summarize, the interpreter fetches words from the input stream one at a time, looks them up in the dictionary, and executes them. If the word is not found in the dictionary, FORTH will attempt to interpret the word as a number, convert it to its binary form, and push it to the stack. What if the word

can't be interpreted as a number? Then FORTH prints the word followed by a question mark and waits for another input line. The rest of the input text is ignored. As it turns out, it is quite possible that FORTH will not be able to convert 2 into a number. Read on.

## DECIMAL - BASE TEN

One of the words in FORTH is **base .** It is a variable which contains the number base which will be used for input-output conversion of numbers. **decimal** is also a word. Here is its definition:

```
: decimal   10  base  ! ;
```

Whenever the interpreter finds **decimal** in the input and executes it, 10 (decimal) is pushed to the stack followed by the address of the variable **base** then 10 is stored at the address which base put on the stack. In other words, when **decimal** is executed, it sets the number base used by the interpreter to ten.

## HEXADECIMAL - BASE SIXTEEN

**hex** is also in the dictionary. It has been defined as

```
: hex   16  base  !  ;
```

When **hex** is executed it sets the number base to sixteen.

Now a mild mind-bender. The definitions of **decimal** and hex listed above assume that at the time they were put in the dictionary the base was ten. If the base at the time they were defined was sixteen, then their definitions would have to be

```
: decimal    0A  base  !  ;
: hex        10  base  !  ;
```

Changing the value of the base changes how strings of digits are interpreted in the input stream, and how bit patterns will be translated on output.

Enter **decimal 14 15 16 hex . . .** and hit the return key. You have an interactive base conversion calculator. Enter some more numbers, change the base (to something as weird as 27, if you wish), then print out the numbers. Try any base you like. When you have had enough of this foolishness, enter **decimal** and hit the return key. You will be on familiar turf again.

 copyright 1983

hit the return key. You will be on familiar turf again.

## BINARY - BASE TWO

Conversion from decimal to binary is certainly a tiresome activity. Obviously FORTH can do it for you. Enter

**decimal 89 2 base ! .**

and hit the return key. FORTH will print out "1011001". If you do a lot of decimal to binary or binary to decimal conversions, you may grow weary of entering **2 base !** all the time, so let's define a word which will set the number base to two.

As mentioned earlier, a definition of a new word to be added to the dictionary begins with a colon and ends with a semicolon. In addition, we must provide a name for the new word. Remember that a name may be any sequence of characters you like. All our word has to do is store a 2 into the variable base .

## CHOOSING NAMES

One of the most important aspects of FORTH programming is choosing good names for new words. One good rule is to focus on what a word does rather than how it does it. We could define our new word as

**: 2base! 2 base ! ;**

and it will certainly do what we want. But its name focuses on how the word works rather than what it does. What's a better name? How about the one in this definition?

**: binary 2 base ! ;**

Did you enter this definition and get an error? We set the base to two, remember? And 2 is not a valid number in base two. So, enter **decimal** and try again.

You may put as many spaces as you like between the words you enter. But you must enter at least one space between each FORTH word. For example,

**decimal:binary2base!;**

will not do at all even though it is relatively readable. Spaces are the traffic cops in FORTH; they are the only way the

**copyright 1983** 

interpreter can tell where one word ends and the next word begins. You will have to rid yourself of that horrible BASIC habit of eliminating spaces to save space.

If everything went all right, FORTH should have said "ok". Now enter **words** and you will discover that **binary** is in the dictionary all ready to be used. So, enter something like

**decimal 512 binary .**

and hit the return key.

The freedom you have in FORTH of specifying the base which will control numeric output and input conversion carries with it a responsibility to make absolutely sure that you always know what the current base is; otherwise you will be in for some surprises. Some will be amusing; others will be very painful. If you don't know what the value of base is, set it!

 copyright 1983

# CHAPTER 4

## WHAT CAN I DO WITH IT?

Suppose you want to charge the long distance telephone calls on an obscene phone bill to the people who made the calls. The people, let us suppose, are Adam, Betsy, Carl, and Denise. To charge a 37 cent call to Carl we would like to be able to enter

```
37  Carl
```

and hit enter. The new total owed by Carl should be calculated, saved, and displayed. We will need to define variables to hold the totals being accumulated for each person.

```
variable Adam's
variable Betsy's
variable Carl's
variable Denise's
```

Now we define the entry commands:

```
:  Adam    Adam's    @ + dup . Adam's   !  ;
:  Betsy   Betsy's   @ + dup . Betsy's  !  ;
:  Carl    Carl's    @ + dup . Carl's   !  ;
:  Denise  Denise's  @ + dup . Denise's !  ;
```

but they are rather repetitious. It would be better if we could "factor out" all the common operations and put them into a word such as **NewTotal** and define the commands as

```
:  Adam    Adam's    NewTotal  ;
:  Betsy   Betsy's   NewTotal  ;
:  Carl    Carl's    NewTotal  ;
:  Denise  Denise's  NewTotal  ;
```

Let's see what's involved in defining **NewTotal** .

 copyright 1983

## GLOSSARY ENTRIES

But first, let's really do this right. Before we write the definition for **NewTotal** , let's write a description of what it does. This description is called a "glossary entry". It is a good idea to write a glossary entry for each word you define. Six months from now you may look at **NewTotal** and not have the slightest idea of what it does or how to use it. Here is what the glossary entry for **NewTotal** should look like.

| **WORD** | **VOCABULARY** | **BLOCK** | **STACK EFFECT** |
|---|---|---|---|
| **NewTotal** | **forth** | **0** | **(amt adr -- )** |

Adds "amt" to the value stored at "adr", then prints out the new value stored at "adr".

This entry tells us that **NewTotal** is in the **forth** vocabulary. The "0" in the "BLOCK" column means that we are going to enter this definition from the keyboard which means that we won't be able to make changes to it or even look at it again. The next chapter presents a better way to enter definitions. The stuff in parentheses tells us what the "stack effect" of the word is.

In the entry in the "STACK EFFECT" column, the two dashes indicate the point at which the word executes. Anything on the left of the dashes indicates what values the word expects on the stack, and anything on the right of the dashes indicates what values the word leaves on the stack. In this case, **NewTotal** expects two values to be placed on the stack for it to use. When it finishes executing, it will have removed those two values from the stack. It does not put any new values on the stack. Having specified what **NewTotal** should do, we can turn to writing its definition.

## LOOK, MA! NO VARIABLES!

Obviously, **NewTotal** will need to use the address it receives on the stack twice. So we will have to copy it and save the copy somewhere. We could create a variable for this purpose, but that's considered inelegant in FORTH circles. Storing and fetching costs time as well as memory for the variable (including its name). We might try leaving it on the stack, but then the address and its copy are on top of the number to be added to the variable. We can shuffle things around on the stack with **swap** and other stack manipulation words.

```
: NewTotal   swap over @ + dup . swap ! ;
```

Here is a stack diagram of what happens when this version of **NewTotal** executes. Notice that **amt** and **adr** are already on the stack when it executes.

```
        WORD       STACK ---->

    :   NewTotal   amt adr
        swap       adr amt
        over       adr amt adr
        @          adr amt OldSum
        +          adr NewSum
        dup        adr NewSum NewSum
        .          adr NewSum
        swap       NewSum adr
        !
    ;
```

The problem here is that unless you have had lots of experience with FORTH, this definition of **NewTotal** is virtually unreadable without the aid of a stack diagram. There is another way to get stack values temporarily out of the way that may help things a little bit. It is time to introduce you to the Return Stack.

## THE RETURN STACK

FORTH uses two stacks. They are called "the stack" (technically, the "parameter stack") and the "return stack". The primary function of the return stack is to hold FORTH return addresses and loop parameters, neither of which we have discussed yet. For now, we will look at another use of the return stack: a place to temporarily put numbers that are in the way.

Suppose that you have a value on top of the stack that you wish to use, but there are values below it that you want to do something to first. The value on top can be moved to the return stack with the word **>r** (pronounced "to R") and retrieved with the word **r>** (pronounced "from R"). If you wish to leave this value on the return stack, but have a copy of it put on the parameter stack, use **r@** (pronounced "R fetch"). Obviously, these words must be used with care, else your temporary value on the return stack might be used as a return address and FORTH will probably crash. Every **>r** in a colon definition should be paired with a subsequent **r>** in the same colon definition.

Back to our problem of defining **NewTotal** . We can move the address to the return stack with **>r** , get back a copy of it with **r@** , then use that copy of the address to fetch the value at that address, take the sum, copy it, print out the copy, get the address back from the return stack with **r>** , and finally store

the new sum.

```
   :  NewTotal  >r  r@  @  +  dup  .  r>  !  ;
```

Here is a stack diagram.

```
          WORD        STACK ---->

     :    NewTotal    amt adr
          >r          amt
          r@          amt adr
          @           amt OldSum
          +           NewSum
          dup         NewSum NewSum
          .           NewSum
          r>          NewSum adr
          !
     ;
```

Remember, the definition of **NewTotal** must be entered before you enter the definitions of the commands which use it.

**FOOD FOR THOUGHT**

Here's another problem. A friend comes to you with this one. She bought a computer out of curiosity, then realized if she took it down to her restaurant, she could write off what she paid for it as a tax deduction. What she wants to do is set it by the cash register and use it to add up her customer's checks. She tried to write a program to do it (in BASIC, of course) without any success. Can you help?

The restaurant is busy and has lots of employees. The program should be simple enough to use so that any of them can operate it. You suggest using plain, ordinary English. How about if they step up to the computer and enter

**Total for blt fries and shake is**

and hit return? That's great, she says, but can you do it? Sure. Won't it take a long time? Is three minutes a long time? You're kidding! Not with FORTH.

The basic trick is to have **Total** push a zero to the stack, have the names of food items add their price to the total on the stack, and have **is** print out whatever is on the stack. Words like **for** and **and** shouldn't do anything. So, we start with the following definitions.

```
: Total    0 ;
: is       . ;
: for   ;
: and   ;
```

Not bad so far. What about the names of food items? We could define them this way.

```
: blt    195  +  ;
: fries   75  +  ;
: shake  125  +  ;
```

and define other items the same way. And, except for defining everything on the menu, we're all done. Not bad, huh?

## DEFINING A WORD THAT DEFINES OTHER WORDS

Actually there's a better way to define menu items. Each item on the menu is a name associated with a price and an action to be performed on that price. It would be much more convenient if we could define menu items this way:

```
195  price  blt
125  price  shake
```

and so on. For this to work, we must define **price** in such a way that when it executes it defines a new word. And when **price** defines another word, it should also specify what happens when that word executes. Here's how.

```
: price  create  ,  does>  @  +  ;
```

Enter the definitions of **Total** and **is** given above then enter the definition of **price** then enter

```
195 price blt
125 price shake
Total blt shake is
```

and, Good Grief!, it works. How does **price** do it?

## WHAT DOES does> DO

When the interpreter looks at **195 price blt** it first pushed 195 to the stack, then **price** was executed. When **price** executes, it first executes **create** which gets the next word in the input, **blt** , and adds it to the dictionary. Next, the comma executes

which removes 195 from the stack and places it in the dictionary as part of the definition of **blt** , then **does>** executes.

**does>** waves its magic wand over **blt** so that when **blt** executes, it will first push, to the stack, the address where the 195 was stored, then it will execute the words which follow **does>** in the definition of **price** .

So, when **blt** executes, it fetches the 195 to the stack then adds it to whatever is already on the stack. That's just what we want **blt** to do. Now other menu items can be added to the "program" with very little effort and in a much more obvious way.

## GETTING FANCIER OUTPUT

One thing about these two samples that is not very nice is that we must work with numbers that are whole numbers of pennies. It would be better if these applications printed out things like "$3.57" instead of simply "357". Can we do it? Sure. All we have to do is define a word to be used in place of the dot. Here it is:

```
: $. 0 <# # # ascii . hold #s ascii $ hold #> type ;
```

How does it work? First, it pushes a zero to the stack. This word will print a 16-bit number which is on the stack. However, the words in this definition which do output formatting of a number on the stack assume that it is a 32-bit number. So, the additional zero simply converts the 16-bit number to a 32-bit number. (This only works if the number is positive.)

The word **<#** sets things up to begin creating the printable "picture" of the 32-bit number on the stack. Then, **#** converts the right-most digit (the pennies digit). The next **#** converts the dimes digit, and the phrase **ascii . hold** inserts a period into the output string we are building. Next, **#s** converts all of the remaining digits in the number giving at least one zero, then **ascii $ hold** inserts a dollar sign onto the output string. **#>** cleans up by dropping the 32-bit number remaining on the stack (it is now a zero), and pushes the address of the first character in the output string and the number of characters in the output string to the stack. **type** uses these values to print out the string. **type** does not print any leading or trailing spaces.

Now you can enter the definition of **$.** then re-enter the definitions of **is** and **NewTotal** replacing the dot with **$.** and that's it. "What?" you may be asking. "I have to re-enter the whole definition of **NewTotal** again? Can't I just edit it?" No, because you entered it from the keyboard. There is another, far

more convenient way to do all of this which we will get to in the next chapter.

## USING FANCIER INPUT

Our output is more "professional" looking, but the input is not. Can't we enter things like **3.95** or **400.** and have them interpreted as $3.95 and $400.00, respectively? Partly. Go ahead and enter them, it's "ok" with FORTH. Now enter a dot. A zero? Enter another dot. There's the number you entered. What's that extra zero doing on top of it?

Any number you enter with a decimal point in it is interpreted by FORTH as a "double" number; as a 32-bit number which the interpreter simply pushes to the stack. The zero is just the "high-order" 16 bits of the 32-bit number. Enter **4000000.** then enter two dots. Weird. Enter **4000000.** again then enter **d.** and things will look better. **d.** does the same thing the dot does except that it interprets the top two 16-bit numbers on the stack as being a single 32-bit number which it removes, converts to a string, and prints it out.

## DOUBLE NUMBERS

The interpreter will interpret a number in the input as being a 16-bit (i.e., "single") number if it is not "punctuated" and as a 32-bit (i.e., "double") number if it is "punctuated". The following are interpreted as single precision numbers:

**0 -13000 13000**

The following are interpreted as double precision numbers:

**0.0 0. .0 -3556.22 .4999**

There may be more than one punctuation character in the number and other punctuation characters besides the period are

**, / - :**

but the dash may not precede the first digit. A leading dash specifies a negative number. Consequently, the following are interpreted as double numbers.

**12:29:15 7/16/83 343-34-3434 555-1212 -23.56**

The last one is interpreted as a negative double number.

Notice, however, that the interpreter converts all of the following to the same internal 32-bit, binary representation.

**100.4 10.04 .1004 1004.**

The only difference is that the variable **dpl** is set to equal the location of the rightmost punctuation character. After "1004." is interpreted, **dpl** equals zero, after "10.04" is interpreted, **dpl** equals two and so on. If **dpl** is negative, then no punctuation character was encountered and the number was interpreted as a 16-bit number.

If you enter **123.45** the interpreter will push a 32-bit number to the stack. Now enter **d.** and see what is printed. Enter **1234.5 d.** and see what is printed. They are the same. And there is no decimal point. What's the difference? The value left in **dpl** after each one was interpreted. This opens the possibility of writing a word which will scale a double number according to the value of **dpl** . Here it is.

```
: scale   dpl @ 0 3 within not
          abort" Entry is out of range."
          drop 2 dpl @ ?do base @ * loop ;
```

(Don't bother to enter this. There are too many possibilities for mistakes and we are almost to the previously mentioned next chapter.) The first line fetches the value in **dpl** and checks to see if it is equal to or greater than zero and less than three. **within** removes **dpl** and the zero and the three from the stack and leaves a "flag". If the flag is zero (false), then **dpl** is not within the specified range; if the flag is -1 (true), then it is. **not** inverts the flag so that it is now true if **dpl** is not within the specified range.

**abort"** removes the flag, and if it is true, it prints out the string which follows it and executes **quit** which terminates execution of **scale** and returns control back to the keyboard. If the flag is false, execution continues by dropping the high-order part of the 32-bit number leaving a single number. A loop then multiplies the number by the current base the proper number of times. (You may want to enter prices in base two.)

Once again, we can modify the definition of **NewTotal** so that we can make phone call entries with **1.16 Carl** and **4. Denise** .

```
: NewTotal  >r scale r@ @ + dup $. r> ! ;
```

We can use **scale** in our definition of **price** as follows:

```
: price create scale , does> @ + ;
```

and enter menu items with **1.95 price blt .**

**IT'S THE PHONE AGAIN**

We can use the magic of **does>** to make our phone bill application even better. In this case, we want to be able to define a number of people and associate with each one a name, a running total, and an action. Here's how.

```
: caller  create 0 , does> NewTotal ;
caller Adam
caller Betsy
caller Carl
caller Denise
```

When **caller** executes, it adds a word to the dictionary, stores a zero with it, then **does>** waves its magic wand so that when the new word executes, **Adam** , for example, the address where the zero was initially stored is pushed to the stack and **NewTotal** is called. This way we don't have to define both **Adam** and his variable.

But what if we enter a bunch of phone charges, decide we are using the wrong bill and want to start over? With the first way of doing it we can simply reset all the variables to zero with a sequence like **0 Adam's !** and start over. How do we do that with this new way? Wow, look at the time. We'd better be getting on to the next chapter.

 copyright 1983

# CHAPTER 5

## HOW DO I SAVE AND EDIT MY DEFINITIONS?

The point has been made that entering definitions from the keyboard has serious limitations. We cannot look at the definitions we have entered when we have forgotten how the entered words were defined. Even worse, we can't modify those definitions. There should be a better way, and there is. Like most other languages, we can save our definitions on disk which gives a permanent record of what we have done and allows us to change things if the need arises. FORTH, however, views the disk a bit differently than other languages. There are no files. ("What do you mean, there are no files! You've got to be kidding!") Since files are the backbone of every other disk operating system, we will probably hear a lot of muttering in the background throughout this chapter. Actually, a good way to tell when you have finally become a good FORTH programmer is discovering that you no longer wish you had files.

### THE FORTH MEETS THE DISK

To FORTH, "the disk" is simply a sequence of "blocks" of data. A block consists of 1024 bytes of data. Each block has a number and the blocks are mapped onto "the disk" in an obvious way.

Block 0 refers to the first 1024 bytes of data on the first track of the disk in the first disk drive. Block 1 refers to the second 1024 bytes of data on the first track of the disk in the first disk drive. Block 152 (or 493 or 615 or whatever, depending on the capacity of the disk) refers to the last 1024 bytes of data on the last track of the disk in the first drive. Block 153 refers to the first 1024 bytes of data on the first track of the disk in the second disk drive. And on it goes.

In order for a program to work on a block of data, that block must first be read into memory. FORTH maintains buffers to hold blocks which have been read in from the disk. In addition to 1024 bytes of data, a buffer has two additional bytes to hold the number of the block which is currently in the buffer, and two null bytes following the data bytes to mark the end of the buffer. As supplied, eFORTH maintains four buffers, and this number can be adjusted, but there must be at least two buffers.

 copyright 1983

A block of data is accessed with the word **block** which expects the number of the requested block to be on the top of the stack. **block** searches through the buffers to see if the requested block is already in memory. If it is, **block** returns the address of the first data byte in the buffer where it found the block (by replacing the block number with the address). If the block is not in a buffer, a buffer is selected, and the block is read from the disk into the buffer, then, as before, the address of the first data byte is returned on the stack.

If you read a block into a buffer and make changes to the data in the block (with editing commands, for example), the buffer is marked as "updated" (the word "dirty" is used in some circles). If that buffer is later required for a requested block which is not in memory, the updated block in that buffer is written out to the disk, then the requested block is read into the buffer.

You can force the writing of all updated buffers to the disk by executing **flush** . You can prevent the writing of all updated buffers by executing **empty-buffers** but all changes made to every block currently in the buffers will be lost.

This scheme is quite simple and powerful, and it is the foundation of most disk operating systems. If you absolutely must have a file system FORTH gives you the basic tools you need to write one.

## PUTTING TEXT ON A BLOCK

We can interpret the contents of a block as being any type of data we like having any kind of structure we like. An obvious possibility is to view the 1024 bytes on a block as being 1024 characters of text. Text is typically organized into lines with some number of characters on each line. A simple scheme is to suppose that each line has some exact number of characters on it, say 64. 64 goes into 1024 exactly 16 times, so we can view a block which has text on it as containing 16 lines of text with 64 characters on each line.

Most FORTH editors make these assumptions, and the eFORTH editor is no exception. Let's use the editor to save the applications we developed in the last chapter starting with the phone bill application.

## THE CURRENT BLOCK

We first need to find a block that isn't being used for anything. Enter **10 list** and hit return. **list** specifies the current block by setting the variable **scr** equal to the block number.

No, that block has stuff on it. Gee, whiz! There's the definition of **list** and it's only three lines long! Sure enough, it stores the block number into **scr** . Notice that line 0 contains a "comment" which briefly describes what is on the block and has a date on it. (The date is automatically put there by the eFORTH editor every time a change is made to the block.) It also has the initials of the person who made the last modification to the block. If you want to see your initials up there in the bright lights of line 0 enter

**I'm cee**

except replace my initials with yours.

Putting a comment on line 0 is a common convention (not a requirement) which helps to document what is on a disk. The word **index** , which is defined on this block, takes advantage of this convention. For example, enter **48 60 index** and hit return. (If you are using eFORTH and followed the directions for setting it up on your computer, you should have at least 85 blocks available.) **index** will print out line 0 on blocks 48 through 59. Block 54 appears to be empty. Let's list it just to be sure. All 16 lines appear to be blank so let's use it. However, there may be junk on this block which **list** doesn't show us. To be absolutely sure that the block is clean for editing, enter **wipe** and hit return.

Oh dear. Where's **wipe** ? It's in the **editor** vocabulary, so enter editor wipe and it should be "ok".

## THE CURRENT LINE

All editing commands operate on the "current" line. We specify that line 0 is the current line by entering **0 t** . Try it. This command also prints the current line. Notice the caret at the beginning of the line. This is the "cursor" and it indicates the current cursor position. More on this later.

Let's put a comment on this line which indicates what's on the block.

**p ( NewTotal caller NamesOfCallers**

A comment begins with a left parenthesis, which must be followed immediately by a space, and ends with a right parenthesis. We do not include the right parenthesis because the editor will automatically put it on the line for us. (Later, when we give this block to the FORTH interpreter, everything inside the parentheses will be ignored.)

Enter l (lower case "L") and hit return. This command always lists the current block. Now enter the following lines:

```
u : $.  ( n -- )
u      0 <#  # #  ascii . hold  #s  ascii $ hold  #> type ;
u : scale  ( d -- n )
u      dpl @   0 3 within  not
u      abort" Entry is out of range."
u      drop  2 dpl @  ?do base @ * loop ;
u : NewTotal ( amt adr -- )
u      >r  scale  r@ @ +  dup $.  r> ! ;
u : caller ( -- ) create 0 , does> NewTotal ;
u caller Adam ( amt -- )
u caller Betsy ( amt -- )
u caller Carl ( amt -- )
u caller Denise ( amt -- )
```

then use the l command to look at the block.

The u command first moves all the lines below the current line down one line. Line 15 is rolled off the bottom and lost. The line "under" the current line is cleared then it becomes the current line. Then the command puts the text which follows it onto the current line.

This block is a little crowded, but we'll take care of that later.

## REPLACING AND DELETING LINES

Did you make a mistake that needs to be corrected? Make the line with the mistake on it the current line. Now use the p command to replace it. Is there an extra line you just want to get rid of? Or did Denise move away? Make that line the current line then enter p followed by two spaces and hit return. The line will be blanked. Or make the line to be eliminated the current line, then enter x and hit return. The current line will be deleted and all lines below it will be moved up. Line 15 is filled with blanks.

Do you need to shuffle some lines around? For example, you might have put the definition of **caller** on a line above the definition of **NewTotal** (which you can't do because **NewTotal** has to be defined before you use it in the definition of **caller** .) For practice, let's move the line with **caller Denise** on it to the line below the line with **caller Adam** on it. Actually, we want to insert it at that point. Make Denise's line the current line then enter **x** and hit return. Now make Adam's line the current line and enter **u** followed immediately by return. Now look at the block.

## THE INSERT BUFFER

The editor maintains an "insert" buffer. Any text which follows **p** and **u** is placed into the insert buffer, and any line deleted with **x** is also placed into the insert buffer. If the **u** or **p** command is entered and followed immediately with a return, it uses the text in the insert buffer rather than what follows it on the line you entered.

## STRING EDITING COMMANDS

The string editing commands include commands to find, delete, and insert strings. Make line 0 the current line, then enter

**f Adam**

and hit return. Notice that the cursor (the caret) is positioned immediately to the right of "Adam". Now enter **f** followed immediately by return. That error message means that "Adam" wasn't found. The f command starts searching at the current cursor position and continues until an occurence of the string is found or until the end of the block is reached in which case it reports that it didn't find the string. When a string is not found, the cursor remains where it was before the string was searched for.

## THE FIND BUFFER

The editor also maintains a "find buffer". Any string which follows **f** is placed into the find buffer. Whenever **f** is followed immediately by a return (or just one space) it will search for the string which is already in the find buffer.

Let's replace all the occurrences of "caller" with "Caller". Make line 0 the current line, then enter

**f caller**

and hit return. When the first line with "caller" on it is printed, enter

**r Caller**

and hit return. Now enter **f** followed immediately by hitting return. The line with the next instance of "caller" on it will be displayed. Enter **r** followed immediately by hitting return. Continue until the editor reports that there are no more instances of "caller" on the screen, then list the screen.

Once a string is found, it can be erased with the **e** command. The **d** command combines the actions of **f** and **e** . It will search for the string which follows it (or which is in the find buffer if it is immediately followed with return) then erase it.

Once the cursor has been positioned with one of the commands that does searching, a string can be inserted at that point with the **i** command. It will either insert the string which follows it at the point where the cursor is positioned or it will insert the string already in the insert buffer if **i** is followed immediately by a return.

The **till** command deletes everything between the cursor and the string which follows it (or is in the find buffer if **till** is immediately followed by a return). **till** does not search beyond the current line.

## HOW TO INTERPRET A BLOCK

Now that we have some FORTH words on a block, we want to have the interpreter interpret what's on the block instead of stuff we enter at the keyboard. How is that done?

First, let's protect ourselves by entering **flush** and hitting return. This will write the block we just edited to the disk. Now, if something goes wrong and we crash, all the editing we did will not be lost. Next, let's get rid of the words we have entered from the keyboard so far. Enter **empty** and hit return. Every word which has been defined since the computer was turned on and FORTH started running will be erased from the dictionary. Now enter **54 load** and hit return. Once the interpreter sees **load** it will stop interpreting the line we typed in and go and interpret the text on block 54. The interpreter will interpret

everything on the block until it reaches the end of the block or until something happens to keep it from reaching the end of the block. Once it finishes doing that it will continue interpreting any text which follows **load** . Since there isn't any, it will just say "ok" and wait for us to enter another line.

## ERRORS WHILE LOADING

It is quite possible, of course, that loading did not go "ok". Typically, this happens when a word has been mis-spelled. As usual, FORTH will print out the word followed by a question mark.

Let's make this happen and see how to correct it. Make line 0 the current line, then enter

**f Adam**

and hit return. Then enter

**f Caller**

and hit return. Then enter

**r caller**

and hit return. This block defines **Caller** but it does not define **caller** because eFORTH does not believe that "c" is the same as "C". Now enter **empty 54 load** and hit return. When the interpreter finds **caller** on the block, it will not find it in the dictionary or be able to convert it to a number, so it gives up and tells you it couldn't do anything with it. Now enter **v** and hit return. The cursor will be positioned immediately after the offending word. Since you know what is wrong with it, fix it by entering

**r Caller**

then enter **empty 54 load** and hit return.

## ANSWERING THE PHONE PROBLEM

Remember the problem we left you hanging with at the end of the previous chapter? Here we have the solution. Simply remove the phone application words from the dictionary with **empty** or **forget** and load them again. Everything will be properly initialized.

## BACK TO THE RESTAURANT

Let's put the restaurant application words onto blocks. This is a test, so you're on your own except for the following suggestions. Make sure each block you use is empty, and **wipe** it before putting anything on it. Put the definitions of **Total and for is** and **price** onto block 58. Put the things defined with **price** onto block 59. Now put the following lines onto block 57:

```
58 load
59 load
exit
```

**exit** will cause the interpreter to stop interpreting the block. Since there isn't much on the block, we put it there so the interpreter won't waste time looking for words in all that empty space.

Block 57 is called a "load" block. All it does is control the loading of all of the blocks which contain words related to an application. All we have to do to load all of our restaurant application words is enter **57 load** and hit return.

## HOW DID YOU DO?

Ready? Did you start a comment on line 0 of all three blocks including block 57? You'd better, or the interpreter will find your initials (or the date) and not know what to do with them. Did you **flush** your work to the disk? All set? Enter **empty 57 load** and sit back. Rats! When we executed **empty** we removed **scale** and **$.** from the dictionary. We will have to put them back by loading block 54 again. However, it was real handy to use **empty** when we encountered an error, fixed it, and re-loaded. What can we do?

Try this. Enter

```
: ***** ;
```

then enter **57 list** to make block 57 the current block, then enter **0 t** to make line 0 the current line, then use the **u** command as follows:

```
u forget *****              : ***** ;
```

Now when we load block 57 it will **forget ******* and everything that was added to the dictionary before the error. Then ********* is redefined so that we are all set up to do the same thing in case we run into another mistake. Now load block 57. If you get an

error, fix it, and load block 57 again. Simple. When we enter **words** this word is real easy to see, and we can tell where, in the list of words, our application begins.

Once we have successfully loaded all the words in our application and we are satisfied that they are working correctly, we can erase line 1 on block 54. It's just a program development tool.

## THE ANSWERS, PLEASE

At the end of this chapter there is a listing which shows what your blocks should look like at this point. The vertical bar on block 57 is a special eFORTH word which tells the interpreter to skip the remainder of the line. So, we follow it with at least one space then use the rest of the line for comments. Are there any questions?

## ELIMINATING CRAMPS

Block 54 is very crowded. List it. Now enter **n l** and hit return. The **n** command makes the "next" block the current block (in this case, block 55). **wipe** it then enter **b l** which makes the current block go "back" one block. Notice the line that the first **caller** is defined on. It should be line 10. Now enter **n** to make block 55 the current block then enter **1 t** to make line 1 the current line. Now enter

**54 10 g**

and hit return. This line "gets" line 10 on block 54 and inserts it under the current line of the current block. So, we have copied line 10 on block 54 to line 2 on block 55. Enter

**54 11 3 gets**

which will copy 3 lines beginning with line 11 on block 54 to the three lines under the current line on the current block. Notice that this command pushed the bottom three lines of block 55 off of the block. They are gone forever.

Now enter **b 10 t** then enter **x x x x** which will erase the lines on block 54 that were copied to block 55. We have effectively moved them from block 54 to block 55.

One minor problem remains. Now, when we load block 54, the words on block 55 are not loaded as well. They should be. A quick solution is to make line 14 (or some line near the bottom) of block 54 the current line, then enter

**p -->**

and hit return. This puts the "arrow" on that line. When the arrow is executed by the interpreter, it stops loading of the current block and forces loading to continue with the next block. Any text on block 54 which follows the arrow will be ignored by the interpreter.

## BLOCK EDITING COMMANDS

Entire blocks can be moved around with the copy command. For example, **54 55 copy** will copy the entire contents of block 54 onto block 55. Any data previously on block 55 will be destroyed.

A sequence of blocks can be copied with the **copies** command. Entering **54 84 3 copies** will copy block 54 to block 84, block 55 to block 85, and block 56 to block 86.

A block that is not the current editing block can be wiped clean with the **clear** command. Entering **54 clear** will fill block 54 with spaces. Be careful! **clear** does not ask you if you are sure. And if you enter **20 clear** thinking the interpreter is in base ten, you may be surprised to discover it was in base sixteen and you have destroyed valuable data on block 32.

These words obviously provide other methods for eliminating cramps.

## DOCUMENTING YOUR APPLICATION

Once everything is working the way it should, you can print a listing of the blocks which contain the source code of the words in your application. For example, the listing in Appendix C was printed by entering

**print 0 72 show ok**

The word **print** is defined to redirect all output generated by any words which appear between it and **ok** to the printer.

The version of **show** which comes with eFORTH only prints three blocks per page. If you have a printer which can be configured to print 132 characters on a line, there is an alternate version you can use which prints six blocks to a page. It's on block 61.

Suppose you make a change to the source on block 55. To get an updated listing, you only have to enter **55 listing** which will print out the page which contains block 55. You do not have to print a new listing for the entire application or the entire disk. The same block will always fall on the same place on the same page.

 copyright 1983

```
Block #   54
 0 ( NewTotal  Caller NamesOfCallers          12:47pm cee 23jan84 )
 1 : $. ( n -- )
 2     0 <# # # ascii . hold #s ascii $ hold #> type ;
 3 : scale ( d -- n )
 4     dpl @  0 3 within not
 5     abort" Entry is out of range."
 6     drop 2 dpl @ ?do base @ * loop ;
 7 : NewTotal ( amt adr -- )
 8     >r scale r@ @ + dup $. r> ! ;
 9 : Caller ( -- ) create 0 , does> NewTotal ;
10 Caller Adam ( amt -- )
11 Caller Betsy ( amt -- )
12 Caller Carl ( amt -- )
13 Caller Denise ( amt -- )
14
15
```

```
Block   55
```

```
Block #   56
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
Block   57
  ( Menu Application Load Block

     58 load ( Total is for and price
     59 load ( prices

  exit
```

```
Block #   58
 0 ( Total and for is price                   12:47pm cee 23jan84 )
 1
 2 : Total ( -- 0 ) 0 ;
 3 : is  ( n -- )  $. ;
 4 : for  ( -- ) ;
 5 : and  ( -- ) ;
 6 : price ( n -- )   scale create , does> @ + ;
 7
 8
 9
10
11
12
13
14
15
```

```
Block   59
  ( prices

  1.95 price blt
  1.25 price shake
   .75 price fries

  exit
```

# CHAPTER 6

## DOES FORTH HAVE WHAT COUNTS?

FORTH implementations generally do not come with words which have as their sole purpose the declaration and manipulation of arrays as a separate data type. As usual, you may add your own if the need arises. And you are surely thinking that the need will inevitably arise. Of course it will, but the creation and manipulation of arrays is quite easy with the FORTH tools already at hand.

### LET ME COUNT THE A's

A frequent application for which arrays are used is to count things when there are a lot of the things to be counted. The trick is to assign a number to each of the things, and use that number as an index into the array. For example, let's count the characters on a block and find out how many a's and other characters there are on the block.

How many different characters are there? Current FORTH standards specify that the internal representation of characters shall be the ASCII character codes. In the ASCII character set there are 96 printable characters and 32 control codes. The ASCII codes start with zero and go as high as 127. So let's just use the internal ASCII code of a character as the index into the array. This means that our array will have to have 128 elements.

How large should each element be? Since we will also be counting spaces, and since a block may be completely blank, we may have to count as many as 1024 spaces. Hence, at the least, each element of the array will have to be large enough to hold a 16-bit integer. We will have to reserve two bytes for each element. Here's how to do it.

```
create Letters  256 allot
```

This line creates a word with the name **Letters** , then 256 bytes (128 elements at two bytes each) are reserved which can be accessed using the word **Letters** . When a word defined with **create** is executed, it simply pushes an address to the stack. In the case of **Letters** , this will be the address of the first byte of the 256 which were allotted to **Letters** . Now, for any given

ASCII code, we can get the address of its element in the array by multiplying its ASCII code by two (because we are using two bytes for each element) then add that result to the address returned by **Letters** . Here's the definition of a word which does this.

```
: letter ( c -- adr ) 2* Letters + ;
```

Once we have the address of the element which corresponds to a character what do we do with it? Just add one to the count which is already stored there.

```
: CountOne ( c -- )  letter 1 swap +! ;
```

To count the characters on a block, we need to get the ASCII code for each of the 1024 characters on a block and pass it to **CountOne** to operate on. How do we do that?

Given the number of the block we want to process, we can use **block** to get the address of the first byte (which holds the first character) on that block. Adding one to that address gives the address of the second byte, and so forth. The standard way to do this sort of thing is to use a loop structure which will execute 1024 times and use the loop index to get each character on the block. Here's how this is done in FORTH.

```
: Count ( blk -- )
     Letters 256 erase ( initialize elements to 0 )
     scr !             ( save the block number )
     1024 0 do         ( begin the loop )
          scr @ block  ( get the block address )
          i +          ( add the loop index to it )
          c@           ( get ASCII code for i-th character )
          CountOne     ( process it )
     loop ;            ( do it again )
```

The first line of this definition presets every element in the array to zero. For convenience, the block number is stored in the user variable **scr** which is a "side-effect" of executing **Count** which you may not like. In a moment we will see how to avoid it.

### HOW DO LOOPS WORK?

For now, let's consider what happens when the sequence **1024 0 do** executes. These words set up the execution of a loop in FORTH. The two numbers are pushed to the stack, as usual, then **do** removes them, fiddles with them slightly, then puts them on the return stack. The zero becomes the initial value of the

loop index which means that the first time through the loop, the word i , which returns the current loop index, will return a zero. Each time the word **loop** executes, the index is incremented by one, then it is compared to the loop "limit" which in this case is 1024. As soon as the index equals the limit, the loop is terminated, and execution continues with the word which follows **loop** . So, the last time the loop executes, i returns 1023. In this case, the "body" of the loop consists of all the words between **do** and **loop** and they are the words which are executed each time through the loop. The comments indicate what they do.

## DO THE I's HAVE IT?

Obviously, reporting the number of times each character appears in a block requires another loop. This time we must loop through the array and print the contents of each element. Let's think about what has to be done to process one character.

We should at least print out the character and its count. To avoid formatting problems, let's just print one character per line. We need a word, then, which will start a new line, print the character, then print its count.

```
: ReportOne ( c -- )   cr   dup emit   letter @ . ;
```

Now we need a loop which goes through all the characters and calls **ReportOne** for each one. To make it easy, let's not report the counts of control codes. This means, though, that our loop should not start with an initial index of zero. The first 32 ASCII codes (0 through 31) represent control codes which are not printable characters on most devices. So the first value returned by i should be 32 and the last should be 127.

```
: Report ( -- ) 128 32 do i ReportOne loop ;
```

Notice that the specified limit is 128 instead of 127. Recall that **loop** adds one to the index and if it then equals the limit the loop is terminated. Hence, the last time the body of the loop executes, i returns 127.

 copyright 1983

## CAN I MAKE IT RUN FASTER?

Usually when a program is running too slowly, the culprit is a loop which executes a large number of times. The best way to speed things up is to try and cut down the time it takes the body of the loop to execute. For example, suppose the body of the loop in **Count** takes five seconds to execute. Since the body of that loop executes 1024 times, shaving just one second off of the time of the body of the loop will result in a savings of 1024 seconds each time **Count** executes. It so happens that **Count** is not coded very efficiently. Notice that the address of the block is calculated each time through the loop. It would be much more efficient to calculate that address just once before the loop begins, save it somewhere, and grab a copy of it each time through the loop. In fact it would be even better if we didn't have to add the value of the index to that address. Why not calculate that address and have it be the initial index? Then each time through the loop, the index will be automatically incremented to become the address of the next character on the block.

```
: Count ( -- )
    Letters 256 erase   ( initialize elements to 0 )
    block               ( adr of 1st char - initial index )
    dup 1024 +          ( adr+1 of last char - the limit )
    swap                ( put them in the right order )
    do                  ( begin the loop )
         i              ( address of current character )
         c@             ( get the character )
         CountOne       ( process it )
    loop ;              ( do it again )
```

Notice that the body of the loop in this version contains far fewer operations. More work has to be done before entering the loop, but that work is done only once instead of 1024 times. Notice one more thing. This version avoids the side-effect of the earlier version: it does not change the contents of **scr** .

## DON'T GO OUT OF BOUNDS

There is the possibility of disaster in our counting program. The value of a byte, after all, can be as high as 255, and there could well be a byte on a block which is greater than 127. What would happen? Clearly, **letter** would return an address to something which is not in the **Letters** array. Consequently, **CountOne** would increment something that probably should not be incremented. What can be done to avoid this problem?

Other languages usually check that an index into an array is within the declared dimensions of the array. However, this checking takes additional time, and it is done whether you want it to be or not. FORTH leaves it to you. It is up to you to decide whether this check should be performed, and, if you decide it should be, what to do when an index into an array is out of bounds. This is obviously a case when we should check. Now, what should be done when we find a byte which is greater than 127?

Two strategies come to mind. The first is the strategy used by other languages: abort the program. We can define a word such as

```
: ?bounds ( c -- )  0 128 within not
     abort" index out of bounds." ;
```

and change the definition of **CountOne** as follows.

```
: CountOne ( c -- ) dup ?bounds letter 1 swap +! ;
```

The other stragegy is to continue processing. However, we will have to decide what to do with bytes greater than 127. We can either ignore them, or expand the size of **Letters** so we can count them, or we can subract 128 from them and process them normally. It's up to you, and it depends on what you are trying to do.

**WHAT'S YOUR SINE?**

Another important use of arrays is the creation of tables of constant data such as a tax table or other data that seldom, if ever, changes. For example, it is a simple matter to create a table of sines and use the angle as an index into the table to get the sine for that angle. Wait a minute, you may be thinking, that will only work if the angles are whole numbers; you can't use a fraction as an index into an array. That's right. However, in many cases (graphics, for example) eliminating fractions may not result in any noticeable loss of accuracy, and "calculating" a sine will be much faster. If you don't need nine digits of floating point accuracy, why spend precious CPU time extracting them?

Let's create a table which contains the sines for angles from zero to ninety degrees. Here's how.

```
create SineTable
  0000 , 0175 , 0349 , 0523 , 0698 , 0872 , 1045 , 1219 , 1392 ,
  1564 , 1736 , 1908 , 2079 , 2250 , 2419 , 2488 , 2756 , 2924 ,
  3090 , 3256 , 3420 , 3584 , 3746 , 3907 , 4067 , 4226 , 4384 ,
  4540 , 4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 , 5736 ,
  5878 , 6018 , 6157 , 6293 , 6428 , 6561 , 6691 , 6820 , 6947 ,
  7071 , 7193 , 7314 , 7431 , 7547 , 7660 , 7771 , 7880 , 7986 ,
  8090 , 8192 , 8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829 ,
  8910 , 8999 , 9063 , 9135 , 9205 , 9272 , 9336 , 9397 , 9455 ,
  9511 , 9563 , 9613 , 9659 , 9703 , 9744 , 9781 , 9816 , 9848 ,
  9877 , 9903 , 9925 , 9945 , 9962 , 9976 , 9986 , 9994 , 9998 ,
 10000 ,
```

When this FORTH code is interpreted, each number is placed on the stack (as usual), then the comma puts it in the dictionary. The first number is "comma'd" into the address returned when **SineTable** executes. Hence, given an angle on the stack in the range [0,90] we replace it with its sine by executing

```
: sin90 ( 0-90 -- sine )   2* SineTable + @  ;
```

(we must multiply by two because each sine occupies two bytes).

If the angle is in the range of [0,180] degrees, we can get its sine from this same table by reflection.

```
: sin180   ( 0-180 -- sine )
    dup 90 > if 180 swap - then
    2* SineTable + @ ;
```

and, for the first time, we see the FORTH version of the "if-then" structure. As you might suspect, the usage of these words in FORTH is the "reverse" of what it is in other languages.

### IF...THEN

Enter the following definition from the keyboard

```
: IfTest  if ." true" then ." continue" ;
```

then enter **true IfTest** and see what happens, then enter **false IfTest** and see what happens. Notice that in both cases the number is removed from the stack. When execution reaches **if** it pulls the number on top of the stack and tests to see if it is equal to zero. If it is, the words between **if** and **then** are skipped, and execution continues with whatever words follow **then**

. But if the number is non-zero, the words between **if** and **then** are executed.

So, in FORTH, the condition to be tested comes before the **if** and the words to be executed if the condition is "true" (i.e., non-zero) come before the **then** .

**IF...ELSE...THEN**

Can you have an "else" part? Sure, but its position is reversed as well. Try the following.

```
: ElseTest if ." true " else ." false " then ." continue" ;
```

Then test it by entering **true ElseTest** and **false ElseTest** and you should have the idea. Now we can write a word which will handle sines in the full range of 0 to 360 degrees.

```
: sin360 ( 0-360 -- sine )
    dup 180 >  if 180 - sin180 negate  else  sin180  then ;
```

**WHAT DOES YOUR SINE LOOK LIKE?**

Let's at least have the satisfaction of seeing something done with these words. Try this.

```
: stars ( cnt -- )  0 do ascii * emit loop ;
: bar ( sine -- ) cr stars ;
: SineWave ( -- ) 360 0 do i sin360 bar loop ;
```

and the results will be terrible. Why? A full cycle of a sine wave will require printing 360 lines which is six sheets of paper! The obvious solution is to not print one line for each degree. Instead, printing one line for every five or ten degrees should give us the basic "picture". How do we implement the obvious solution? Introducing the fabulous **+loop** which can be used in place of **loop** to increment the index by some value other than one each time through the loop. For example,

```
: SineWave ( -- ) 360 0 do i sin360 bar 10 +loop ;
```

will increment the index by ten each time through the loop which means that **sin360** will be called with values of 0, 10, 20, etc.

The fabulous **+loop** will even let us run through the indices "backwards". For example,

```
: SineWave ( -- ) 0 360 do i sin360 bar -10 +loop ;
```

will call **sin360** with angles of 360, 350, 340, etc. In this case, 360 is the initial index, and 0 is the limit. There is one slight catch when the limit is lower than the initial index and the loop counts "down". The loop will not terminate until the index value runs below the limit (instead of becoming equal to it). For example, enter and execute the following

```
: up 5 0 do i . 1 +loop ;
: down 0 5 do i . -1 +loop ;
```

and notice the difference. The "up" loop will execute five times with the index running from zero to four. The "down" loop will execute six times with the index running from five to zero. Try these again with increments of 2 and -2 and see what happens.

Even with these new versions of **SineWave** the results are still terrible. The reason is that the number of stars printed on a line could be -10,000 or +10,000. That range is a bit out of whack considering that most display devices will handle no more than 80 stars on a line. It's obvious that **bar** should scale things down a bit.

Let's assume that we can get as many as 80 stars on a line. That means that whatever value is given to **bar** should be scaled so that it is in the range (-40,+40). We should then add 40 to the result so that the loop limit is in the range [1,80]. If we divide the sine by 300, the result will be in the desired range. So, we end up with this.

```
: bar ( sine -- ) 300 / 40 + cr stars ;
```

Although this isn't the most sophisticated application of trigonometric functions, it is still interesting to note that our "imprecise", "integer-only", "whole-degrees-only" (add your own pejoratives here) method provides sine values which have greater precision than we need.

## INDEFINITE LOOPS

We have been looking at how to create program structures which are known as "definite" loop structures. Before the loop is entered, you know how many times it will execute. Sometimes, you will want something to happen over and over again, but you

won't know how many times it should happen before you start doing it. This latter type of loop structure is called an "indefinite" loop.

For example, you might want to have the sine of 4000 degrees or -10 degrees. 4000 degrees can be interpreted as going around in a circle (to the right) over 10 times, and -20 degrees can be interpreted as going around in a circle 20 degrees to the left. Consequently, the sine of -20 degrees is equal to the sine of 340 degrees since turning 20 degrees to the left leaves you heading in the same direction as turning 340 degrees to the right. And turning 4000 degrees to the right leaves you heading in the same direction as turning 40 degrees to the right. You just don't get as dizzy.

How might we convert any number of degrees to the equivalent number of degrees within the [0,360] range? If the number of degrees is positive and greater than 360, we can simply subtract 360 until the result is still positive and less than 361. Here's how.

```
: Right360 ( n1 -- n2 )
    begin dup 360 > while 360 - repeat ;
```

If the number on the stack is greater than 360, a true flag is left on the stack. If **while** sees a true flag on the stack (which it removes) the words between **while** and **repeat** will be executed (the body of the loop), then execution goes back to the point marked by **begin** . Notice that the body of a "while" loop might not be executed at all. Additional definitions will allow us to get the correct sine for any number of degrees.

```
: Left360 ( n1 -- n2 )
    begin dup 0< while 360 + repeat ;
: sin ( degrees -- sine )
    dup 0< if Left360 else Right360 then sin360 ;
```

This definition of **sin** is coded in such a way that the word **Left360** is only called with a negative number of degrees. Hence, it will always add 360 to the number passed to it at least once. Here is another way to code it.

```
: Left360 ( n1 -- n2 )
    begin 360 +  dup -1 > until ;
```

The body of this "until" loop structure will always execute at least once, and it will loop until the number is zero or greater.

## SOME ODDS AND ENDS

A few more details about definite loops should be mentioned. Enter these words from the keyboard and try them out.

```
: up 40000 0 do i . 10000 +loop ;
```

Probably not what you expected. Enter **40000 .** and see what is printed. The internal representation of 40,000 is interpreted by FORTH as a negative number. The loop keeps going and adds 10,000 each time until the index overflows and becomes negative, then it keeps on going until adding 10,000 reaches or passes that negative limit. Try this one.

```
: up 0 0 do i . 10000 +loop ;
```

Obviously this behavior of **do** can be most undesirable in some situations. For example, imagine what would happen if you entered **0 stars** . You would have to either hit the reset button or wait until 65,526 stars are printed.

In situations such as **stars** there is a special word, **?do** , which can be used. If you enter

```
: stars ( cnt -- ) 0 ?do ascii * emit loop ;
```

then execute **0 stars** , no stars will be printed; the body of the loop will not be executed. Nothing will be printed if you enter **-5 stars** because **?do** is defined to not execute the body of the loop if the limit is equal to or less than the initial index.

## IT'S TIME TO leave

There are times when a loop should be terminated before it has executed the predetermined number of times. For example, look at the definition of **s** on block 41. This word is designed to search for a string starting at the current block and the blocks which follow it until reaching the block whose number is on the stack. So, if the current block is block 12, entering

```
45 s c/l
```

will search for the string "c/l" on all blocks from 12 to 45. The basic structure of **s** is a loop with an initial index, in this case, of 12, and a loop limit of 45. However, if an occurence of the string is found, the line it is on should be printed out, and execution of **s** should be terminated so that the user can replace that string with something else (or perform some other operation on it). This is what the word **leave** does. If the string is

found, the words between **if** and **then** are executed. When execution finally gets to **leave** it immediately causes the loop to be exited; execution continues with the word which follows **loop** . Notice that the 45 will be left on the stack. This means that entering s and hitting return will resume the search for "c/l".

A similar word, **?leave** , expects a flag on the stack (which it removes), and if the flag is true, it immediately terminates the loop. If the flag is false, execution continues with the word which follows **?leave** .

 **copyright 1983**

# CHAPTER 7

## WHAT'S IN A WORD?

The dictionary begins somewhere in low memory and grows upward as words are added to it. Let's look at some of the details of what is actually put into the dictionary when a word is defined.

When the FORTH interpreter tackles a line such as

**variable 1#**

it finds **variable** in the dictionary and executes it. If you recall, **variable** takes the next word in the input stream and puts it in the dictionary as the name of a word. In this case the word will be a variable and its initial value will be zero.

All defining words ultimately call **create** which puts in the dictionary those elements which are common to every word in the dictionary whether it is a variable, a constant, or a colon definition. These elements are:

1. **The link field,**
2. **The count byte,**
3. **The name of the word,**
4. **The code field,**
5. **The parameter field.**

### THE LINK FIELD

The first field in a dictionary entry is called the "link field". It is the 16-bit address of the count byte of the previous word (in the same vocabulary). The link field of the last word in the vocabulary is zero. By following these links every word in the vocabulary can be examined.

 copyright 1983

## THE NAME FIELD

The count byte together with the characters which comprise the word's name are collectivly called the word's "name field". The lowest 5 bits of the count byte are reserved for the count of characters in the word's name. Hence, a word's name may be up to 31 characters long. The sixth bit of the count byte is not used. The seventh bit is called the "precedence bit". If this bit is set, the word is an "immediate" word. The point of this will be discussed in a moment. Finally, the eighth bit of the count byte is always set. So is the eighth bit of the last character in the word's name. This allows dictionary scanning words to go from one end of the name field to the other.

## THE CODE FIELD

The third field in a dictionary entry is called the "code field". This field contains a 16-bit address. At this address will be found machine code which is to be executed whenever the word is executed.

## THE PARAMETER FIELD

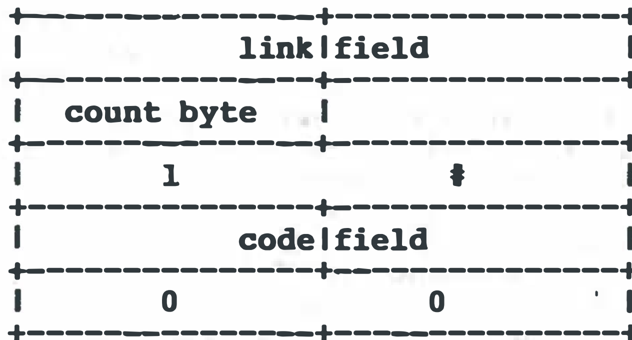
The last field is called the "parameter field". This field can be as short as a single byte or as long as several thousand. The nature of its contents can vary just as widely. In short, the parameter field contains some type of data. The code field points to a machine language program which determines what is done with that data.

## VARIABLES

Once again, take the simple case of a variable. When

**variable l#**

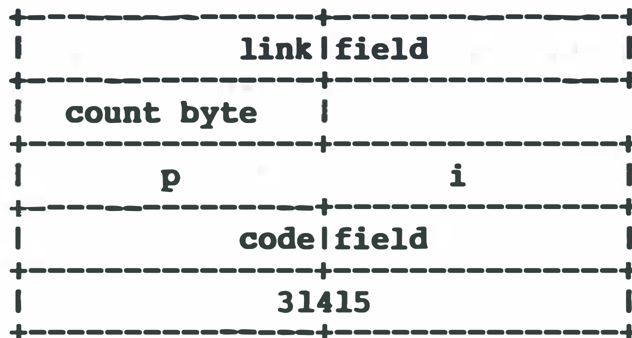
is interpreted, **variable** calls **create** which creates the new word's link field, name field, and reserves space for the code field. **variable** then fills in the code field with the address of a machine language routine which performs the operation associated with variables: pushing the address of the variable's parameter field to the stack. Finally, **variable** reserves two bytes in the dictionary for the new word's parameter field and stores a zero there. Here is a picture of the order of things in memory after a variable is defined.




The word **here** always returns the address of the next free byte in the dictionary. So every time something is compiled into the dictionary the address returned by **here** is advanced.

**CONSTANTS**

When **31415 constant pi** is interpreted, the actions taken are identical to the previous description of what happens when a variable is defined except that the code field of a constant is filled with the address of a different machine language routine; one which pushes the contents of the word's parameter field to the stack (instead of the address of the parameter field).



## COLON DEFINITIONS

Things are a bit more involved for a colon definition. When

**: binary 2 base ! ;**

is interpreted, the colon is first found and executed. The colon calls **create** which adds the appropriate link field and name field to the dictionary and reserves space for the code field. The colon then calls ] which is the word which puts things into the parameter field of the colon definition which is being compiled. The semicolon at the end of a colon definition stops the execution of ] . Next, the code field is filled in with the address of the appropriate machine code. Finally, the new word is added to the dictionary so that other words can use it.

What, you may be wondering, is put into the parameter field of a colon definition? The answer is quite simple. The parameter field of a colon definition is a list of code field addresses. Here is what **binary** looks like.

```
+---------------+---------------+
|           link|field          |
+---------------+---------------+
| count byte    |
+---------------+---------------+
|       b              i        |
+---------------+---------------+
|       n              a        |
+---------------+---------------+
|       r              y        |
+---------------+---------------+
|           code|field          |
+---------------+---------------+
|           cfa of 2            |
+---------------+---------------+
|           cfa of base         |
+---------------+---------------+
|           cfa of !            |
+---------------+---------------+
|           cfa of exit         |
+---------------+---------------+
```

Converting the source text of a colon definition into this list of code field addresses is called "compiling".

## COMPILATION

In FORTH, compiling is a very simple process. The lion's share of this work is done by the word ] which was mentioned earlier. For the sake of convenience, we will refer to this word as "the compiler". Here is its definition:

```
: ]  ( -- )
     true state !  ( indicate that compiling is in process )
     begin         ( start an infinite loop )
       bl word     ( get next word from the input stream )
       find        ( search for it )
       ?dup        ( was it found? )
       if          ( it was found )
         192 <     ( is it an immediate word? )
           if      ( it is not immediate )
             ,     ( compile its code field address )
           else    ( it is immediate )
             execute ( execute it instead of compiling it )
             ?stack  ( check for stack underflow )
           then
       else        ( it wasn't found )
         (number)  ( see if it's a number )
         [compile] literal   ( compile the number )
       then
     again ;       ( process the next word )
```

It should be fairly clear how this word works, but a few comments are in order. Notice that **word** is used to get words from the "input stream". If you entered a definition at the keyboard, then **word** simply gets each word you typed. If you are loading a block with a colon definition on it, then **word** gets each word in that definition off of the block being loaded. **word** is smart enough to know where it is supposed to get the next word.

Next, the word is searched for in the dictionary. If it is not found, **(number)** is called to see if the word can be interpreted as a number. If not, **(number)** prints an error message and aborts the whole process. Otherwise, the number is left on the stack, and **literal** removes it and compiles stuff so that when the word being defined is executed, the number is pushed to the stack. How this is done is discussed in the next chapter.

If the word is found in the dictionary, then that word is either an "immediate" word or it is not. If not, its execution address (code field address) is "compiled" into the dictionary by the comma. This address was left on the stack by **find** and the comma just removes it and sticks it in the dictionary after reserving two bytes for it.

 copyright 1983

If the word is immediate, then it is executed, after which we check for stack underflow.

Since this "compiling loop" is an infinite loop, you may be wondering how the compiling process ever stops. Look at the definition of **binary** again. When the colon executes, it adds **binary** to the dictionary, then calls **]** which compiles the execution address of **2** and **base** and **!** into the parameter field of **binary** (adding two more bytes to its size each time). Finally, **]** fetches the semicolon from the input stream and finds it in the dictionary. What happens now? Does ] compile the semicolon's execution address into the dictionary and go on to the next word and compile its execution address into the dictionary? Hopefully not. The semicolon should, among other things, terminate compilation.

## IMMEDIATE WORDS

The solution is to devise some way of having certain special words, such as the semicolon, executed by the compiler. These words are called immediate words and this explains the existence of the precedence bit in a word's count byte. In short, if a word is an immediate word, its precedence bit is set and the compiler will always execute this word.

The semicolon is an obvious candidate for being an immediate word, and it is. Here is its definition:

```
: ;   ( -- ) compile exit   r> drop   ; immediate
```

When it executes, it compiles **exit** into the dictionary (which shows that ] isn't the only word that can compile things), then it removes a number from the return stack and throws it away which clearly violates the rules for good use of the return stack. Why is this done? This is the way the infinite loop in ] is terminated. We will take up the details in the next chapter.

The word **immediate** which follows the semicolon in the definition of the semicolon simply marks the previously defined word as being an immediate word by setting its precedence bit in its count byte.

## COMPILE TIME AND RUN TIME

Look at the definition of ] again, and notice that **literal** is preceded with **[compile]** . Why? It turns out that **literal** is an immediate word. Consequently, it would normally execute when ] is being compiled, rather than later when ] is executed. Instead, the use of **[compile]** forces **literal** to execute when ] is executed, not when it is compiled. Let's look at this in a little more detail.

The word **literal** is typically used as follows:

```
: line15 ( blk -- adr )  block [ 15 c/l * ] literal + ;
```

Given a block number, this word returns the address of the first character on the last line of that block.

## COMPILE TIME

The phrase "compile time" refers to the time when **line15** is compiled (added to the dictionary). What happens at this time? Once the colon executes and adds an entry for **line15** to the dictionary, it calls ] to start compiling which means that the execution address of **block** is compiled into the parameter field of **line15** . However, the word [ stops compiling and begins interpretation again. This means that 15 is pushed to the stack, then **c/l** executes which pushes 64 to the stack, then * executes which pulls 15 and 64 from the stack and leaves 960 on the stack. Then, ] executes which stops interpretation and resumes compilation. Since **literal** is an immediate word, it executes anyway, and removes 960 from the stack and puts it into the parameter field of **line15** so that when **line15** executes, 960 will be pushed to the stack. Then the plus is compiled, and, finally, the semicolon executes.

## RUN TIME

The phrase "run time" refers to any time when **line15** is executed. What happens at this time? When **line15** is later executed, **block** is executed which leaves an address on the stack, then 960 is pushed to the stack, then + is executed which adds 960 to the buffer address left by **block** and that's it. This distinction between compile time and run time is important. Remember that immediate words are executed at compile time.

It so happens that **[compile]** is also an immediate word, so it executes when ] is compiled. What it does, is get the next word (which, in this case, is **literal** ) from the input stream and compile it. So it is there to prevent **literal** from being executed when ] is compiled. It forces **literal** to be compiled so that it will be part of the run time behavior of ] .

## CODE DEFINITIONS

Many words in the dictionary are not defined with the colon or with **variable** or with **constant** . Many are defined with **code** which allows you to define words in terms of machine code instead of in terms of other words. In this way you may add new "primitive" operations to FORTH including routines which will respond to interrupts and which do other hardware related processing. Sometimes you may want to define words with **code** instead of the colon simply because you want them to execute as fast as possible.

Writing **code** definitions is greatly simplified if an **assembler** vocabulary is available for your particular CPU. The **assembler** vocabulary supplied with eFORTH will be described later.

# CHAPTER 8

## HOW DOES FORTH WORK?

FORTH is most commonly implemented on any given CPU by writing code for that CPU which will simulate an abstract computer here referred to as the "FORTH machine". The only function of the FORTH machine is to execute lists of code field addresses; i.e., the list of code field addresses in the parameter field of a colon definition. In installations of this sort, the only thing done by a cold start routine which gets FORTH running is to initialize the host CPU registers, then to start the simulated FORTH machine (sometimes referred to as the "virtual machine").

If you are using eFORTH, the FORTH machine is running the entire time you are using FORTH. Your CPU is simply executing routines which simulate various operations of the FORTH machine. Since your proficiency as a FORTH programmer will be enhanced by understanding the operation of the FORTH machine, we shall describe it in detail.

### THE FORTH MACHINE'S REGISTERS

The FORTH machine has five registers. They are

1. **IP - The instruction pointer,**
2. **W - The word pointer,**
3. **SP - The pointer to the parameter stack,**
4. **RP - The pointer to the return stack.**
5. **UP - The user pointer**

The stack pointer, SP, always points to the last number which was pushed to the parameter stack. The return stack pointer, RP, always points to the last return address which was pushed to the return stack. The user pointer, UP, points to the origin of the "user variable area". This area makes it possible to implement multi-tasking in FORTH. It is possible to connect two terminals to a computer running FORTH and have two people using FORTH at the same time. Obviously, they should have separate copies of variables such as **base** and others. eFORTH can be expanded to support multi-tasking.

The word pointer, W, points to the code field of the word being executed. The instruction pointer, IP, always points to a location inside some colon definition's parameter field. This location, you recall, contains an execution address. The code field which this execution address points to contains another address which, finally, points to machine code the host CPU can execute. All of this is clearly indirect but, ultimately, quite simple and powerful.

All the FORTH machine has to do is somehow see to it that the machine code ultimately pointed to by the code field address which IP points to is executed and then arrange things so that the next code field address is pointed to by IP and the machine code which it ultimately points to is executed, etc. This basic operation of the FORTH machine is usually implemented in a host CPU machine code routine called NEXT which is also referred to as the "inner interpreter" or "address interpreter". This terminology is intended to distinguish NEXT from the "outer interpreter" or input text interpreter.

## WHO'S NEXT?

Implementations of NEXT are either pre-increment or post-increment depending on which is most efficient to implement on the host CPU. The post-increment version is more common. It assumes that when NEXT is first entered, IP points to the code field address to be processed. This code field address is loaded into the W register, IP is advanced to point to the next code field address, and the word whose execution address is in W is executed.

In pre-increment versions, when NEXT is first entered, IP points to the code field address of the word which was just executed. So, IP is advanced to the next code field address and it is processed as before. W is loaded with the code field address now pointed to by IP and the word whose code field is pointed to by W is executed.

## IMPLEMENTING THE FORTH MACHINE

A FORTH machine is implemented on a given CPU by deciding how to handle the FORTH machine's registers, then writing suitable code for NEXT, DOCOL, EXIT and other primitives required by FORTH. If your FORTH programming will always be restricted to creating colon definitions, you need not be concerned with the details of how the FORTH machine was implemented on your CPU (other than knowing whether it is a pre-increment or

post-increment machine). You only need to know how the FORTH machine works. But, if you intend to write **code** definitions which use the FORTH machine's parameter stack, you must know how to find the pointer to the top of the parameter stack. Is it one of your CPU's registers or does your implementation hold it in memory somewhere?

Here are the answers to these question for eFORTH users.

## THE eFORTH 6809 FORTH MACHINE

Here is a brief discussion of the implementation of the FORTH machine in the 6809 version of eFORTH.

Considerations of efficiency suggest that if it is at all possible, the FORTH machine registers should be implemented with registers on the host CPU. Fortunately, the 6809 has barely enough registers to do this. The 6809 Y register serves as the FORTH machine's IP register, the 6809 X register serves as the FORTH machine's W register, the 6809 U register serves as the FORTH machine's SP register, the 6809 stack pointer serves as the FORTH machine's RP register, and the 6809 DP register serves as the FORTH machine's UP register. Accordingly, NEXT, DOCOL, and EXIT are coded in standard 6809 assembly language as

```
* Y POINTS TO THE CODE FIELD ADDRESS TO BE EXECUTED
NEXT    LDX    ,Y++    POINT W TO CODE FIELD
        JMP    [,X]    EXECUTE CODE

* X POINTS TO THE WORD'S CODE FIELD
DOCOL   PSHS   Y       SAVE IP ON THE RETURN STACK
        LEAY   2,X     POINT IP TO FIRST CODE FIELD ADDRESS
        BRA    NEXT

EXIT    PULS   Y       GET RETURN ADDR INTO IP
        BRA    NEXT
```

Notice that NEXT implements a post-increment version of the FORTH machine. Clearly the 6809 architecture permits an efficient implementation of the FORTH machine.

## THE INTERPRETER

It so happens that the interpreter is itself defined with the colon. In other words, it is a word in the dictionary (its name is **interpret** ), and its parameter field is a list of execution addresses. Here is its definition:

 copyright 1983

```
: interpret ( -- )
    begin
      false state !      ( indicate interpretation is in process )
      -'                 ( get the next word and search for it )
      if                 ( it wasn't found )
        'number @        ( get the execution address of number )
      then
      execute            ( execute the execution adr on the stack )
      ?stack             ( check for stack underflow )
    again ;              ( interpret the next word )
```

This word makes sure that the value in the variable **state** indicates that interpretation is in process (instead of compilation). Then the next word in the input stream is fetched and searched for in the dictionary. Now the code gets a little "tricky". If the word isn't found, the execution address of the word which attempts number conversion is put on the stack. The address of the string to be converted is left under it. If the word is found, its execution address is left on the stack. Either way, by the time things get to **execute** , there is an execution address on the stack of a word to be executed. After the word is executed, the stack is checked, and the process is repeated (another infinite loop).

Suppose the interpreter finds **binary** in the input stream, and that it has been defined. Ultimately, **binary** will find its execution address on the stack, and **interpret** will **execute** it. What happens when **binary** is executed?

Well, the execution address of **execute** is in the parameter field of **interpret** , and when **execute** finishes (by executing **binary** ), FORTH should go on to execute the word whose execution address follows that of **execute** in the parameter field of **interpret** (which happens to be the execution address of **?stack** ).

Obviously, FORTH has to remember where it should go back to. This is the job performed by DOCOL. In the case of **binary** , for example, this code pushes the address in IP to the return stack, then loads IP with the address of the parameter field of **binary** . The words whose execution addresses are in the parameter field of **binary** are executed including **exit** . When **exit** executes, it pulls the address on the return stack and puts it back into IP, and the execution of ] is resumed (by executing **?stack** ). This is why you must be extremely careful when using the return stack.

This is also why the semicolon is coded the way it is. When ] found the semicolon and noticed that it was immediate, it executed it. Since the semicolon is defined with the colon, it first pushes the address in IP to the return stack, then the words in the semicolon's parameter field are executed. The phrase **r> drop** removes the address on the return stack and throws

it away. This exposes the address of a word in the parameter field of the colon (which called ] in the first place.) So, when **exit** at the end of the semicolon executes, it returns to the colon instead of the compiler. This is how the infinite loop is terminated.

# CHAPTER 9

# HOW DOES FORTH COMPILE NUMBERS?

Recall the definition of **binary** which was

**: binary 2 base ! ;**

and recall that 2 is in the dictionary. That fact, that 2 is a defined word, resulted in FORTH compiling the execution address of 2 into the dictionary when **binary** was compiled

## NUMERIC LITERALS

But what does FORTH do when it compiles something like

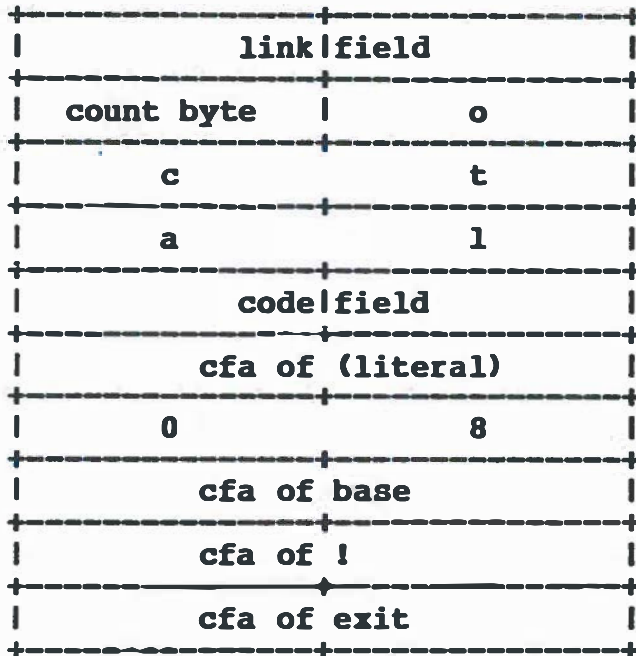
**: octal 8 base ! ;**

when a number such as 8 is not in the dictionary? Unlike 2, 8 has not been defined as a constant. It is referred to as a literal value; instead of being the name of a constant or variable, it is to be interpreted, literally, as the number 8.

Since 8 is not the name of a word in the dictionary, the interpreter cannot compile its execution address into the dictionary because 8 is not the name of anything which has an execution address. FORTH handles this sort of situation by using the special word **(literal)** . Look at the definition of ] again and notice that when it gets a string from the input stream which is not in the dictionary but can be converted to a number, it leaves that number on the stack and calls **literal** which first compiles the execution address of **(literal)** into the dictionary, then it compiles the number into the dictionary. The number is then referred to as an "in-line parameter"; it is compiled in-line with the execution address of the word which will use it. This word is **(literal)** .

**(literal)** is a **code** definition; a machine language primitive. Here is what happens when it executes. Given the current value of the FORTH machine's IP register, **(literal)** can find the literal number which was compiled with it. In the 6809 eFORTH implementation, IP is already pointing two bytes beyond the execution address of **(literal)** . The number which was

compiled in-line with (literal) is at this address. So, (literal) gets it and pushes it to the stack, then advances IP two bytes to skip over the number. This prevents the FORTH MACHINE from interpreting the number as an execution address. In general, NEXT advances IP two bytes each time NEXT is executed because each execution address is two bytes long. However, when (literal) executes, IP is advanced a total of four bytes; two for the execution address of (literal) , and two for the number which follows it.




**BRANCHING**

In-line parameters are also used for the words in FORTH which control program flow. In short, what is compiled into the dictionary when FORTH runs into words such as if , else , then and others? The word if , for example, should cause segments of execution addresses to be skipped over when the condition preceding it is not satisfied. If the condition is satisfied, and execution reaches the else , program flow should skip over the execution addresses compiled between the else and then . How does FORTH handle the compilation of these words?

As it happens, they are immediate words; they are always executed even when FORTH is in the compiling state. What they must ultimately do is compile words into the dictionary which will cause the FORTH machine to skip over segments of execution

addresses. This is handled quite easily by manipulating the contents of the IP register. One such word is **branch** . It always causes a branch, but, we might ask, to where? That depends. The number of execution addresses to be skipped can vary depending on how many words occur between **if** and **else** , for example. Accordingly, **branch** is always followed by an in-line parameter which contains the address to branch to. All **branch** has to do is put this address into the IP register.

**branch** always branches, so **if** must compile some other word into the dictionary; a word which will branch or not depending on what is on the stack. This is the function of **0branch** which will cause a branch if the number on the stack is zero; otherwise execution continues with the execution address which follows the in-line parameter which follows **0branch** .

Let's look at what happens when the interpreter runs into a definition such as

**: 0= if false else true then;**

First, **0=** is added to the dictionary. Since **if** is an immediate word, it is executed. Here is the definition of **if** :

**: if ( -- adr ) compile 0branch here 0 , ; immediate**

The word **immediate** which follows the definition of **if** sets its precedence bit which is what makes **if** an immediate word.

When the phrase **compile 0branch** executes, the execution address of 0branch is compiled into the dictionary (as part of the definition of **0=** remember?). Then **here** is executed which pushes the address of the next free byte in the dictionary to the stack. (This address will later be used by **else** .) Next, the phrase **0 ,** causes a zero to be compiled into the dictionary.

Let's take a closer look at the sequence **here 0 ,** in the definition of **if** . The comma compiles whatever number is on the stack into the dictionary. In this case, a zero. The address which **here** pushed to the stack points to this zero. The zero, of course, is the in-line parameter for **0branch** to use when it executes. Why is a zero used? Because at this point FORTH has no idea of what this in-line parameter should be. Ultimately, it should be an address which **0branch** will place in the IP register. This is called an "unresolved forward reference" and it will have to be "resolved" later.

Remember, **if** should cause the words between it and **else** to be executed if the top of the stack is true (non-zero). If the top of the stack is zero, **if** should cause these words to be skipped and the words between **else** and **then** should be executed.

So, for the time being, the **0branch** in-line parameter is set to zero. It will be changed to the appropriate address when that address is known.

When will that address be known? When **else** executes. Here is the definition of **else** :

```
: else ( adr1 -- adr2 )
    compile branch here 0 ,
    swap here swap ! ; immediate
```

First, **branch** is compiled into the dictionary with zero as a temporary parameter and **here** pushes the address of this parameter to the stack so that **then** can change it to what it should be. The point of this is to branch over the "else" part of the conditional when the "true" part has been executed. Next, this address is swapped with the one left on the stack by **if** (remember?).

We now know what the parameter of **0branch** compiled by **if** should be; it should cause a branch to the next word compiled into the dictionary. The current address of this word is returned by **here** . So, **swap** gets the address of the in-line parameter which follows **0branch** (which was compiled by **if** ) onto the top of the stack. Next, **here** puts the address to which **0branch** should branch onto the stack, but they are in the wrong order, so **swap** fixes this problem, and the correct address finally replaces the zero which was temporarily compiled as the in-line parameter. And the unresolved forward reference created by **if** has been resolved. Notice that the address of the in-line parameter of **branch** compiled by **else** is still on the stack. This unresolved forward reference will be resolved by **then** . Here is the definition of **then** :

```
: then ( adr -- )   here swap ! ; immediate
```

It just resolves the forward reference at the address on the stack.

eFORTH provides tools which make it very easy to create and resolve forward references. Here are better definitions of the program structuring words.

```
: if    system compile 0branch  forward ;  immediate
: else  system compile branch  forward
              swap resolve ; immediate
: then  system resolve ;  immediate
```

Notice that the **system** vocabulary is specified because some of the words in these definitions are in that vocabulary.

When **0=** is finally compiled here is what it looks like.

```
         +---------------+---------------+
         |           link|field          |
         +---------------+---------------+
         |  count byte   |
         +---------------+---------------+
         |       0               =       |
         +---------------+---------------+
address  |           code|field          |
         +---------------+---------------+
 1000    |         cfa of Obranch        |
         +---------------+---------------+
 1002    |             1010              |
         +---------------+---------------+
 1004    |         cfa of false          |
         +---------------+---------------+
 1006    |         cfa of branch         |
         +---------------+---------------+
 1008    |             1012              |
         +---------------+---------------+
 1010    |         cfa of true           |
         +---------------+---------------+
 1012    |         cfa of exit           |
         +---------------+---------------+
```

**WHEN if COMPILES**

Let's consider what happens when the definition of **if** is compiled. First, the colon is fetched from the input stream and executed creating the name, link, and code fields for **if** . Then FORTH is put into the compilation state and the execution address of **compile** is compiled into the dictionary. The same thing happens to **Obranch** and **forward** . The semicolon terminates compilation and compiles **exit** at the end of the list of execution addresses. **immediate** is executed which sets its precedence bit. Nothing extraordinary about it at all. All the magic occurs when **if** executes, not when it is compiled.

**HOW compile WORKS**

A brief word should be said about the behavior of **compile** when it executes. It is another word which expects an in-line parameter. In the definition of **if** , that in-line parameter should be the execution address of **Obranch** and, indeed, when **if** is compiled, the execution address of **Obranch** is compiled

immediately after the execution address of **compile** . When **if** is executed, the FORTH machine will eventually reach the execution address of **compile** and execute it. What happens?

Here is the definition of **compile** :

```
: compile r> dup @ , 2+ >r ;
```

Notice that this definition appears at first glance to satisfy the rule which requires that every **>r** be balanced with a **r>** in the same definition. However, they are backwards. Instead of pushing something to the return stack with **>r** and getting it back with **r>** , **compile** pulls a number from the return stack, uses it to fetch something to the stack, increments it by two, then replaces it. What number is on the return stack when **compile** executes, and what is the purpose of this apparently "illegal" use of **r>** and **>r** ?

The number on the return stack is the return address saved when **compile** was called. This return address points to a location in the parameter field of **if** . Assuming a post-increment implementation of the FORTH MACHINE, the return address on the return stack will be pointing two bytes beyond the execution address of **compile** ; that is, pointing to the execution address of **0branch** . The job of **compile** is to put the execution address of whatever word follows it into the dictionary. Whenever **compile** is executed, the address on the return stack will point to the execution address which **compile** is to put into the dictionary. Consequently, **r>** is used to fetch this address. It is duplicated, the copy is used to get the execution address to be compiled with the comma, then the return address is incremented by two to skip over the execution address to be compiled by **compile** .

In the case of the **compile** in the definition of **if** , when **compile** executes, the return address points to the execution address of **0branch** which immediately follows the execution address of **compile** in the parameter field of **if** . **compile** uses this address to fetch the execution address of **0branch** to the stack and uses the comma to compile it into the parameter field of whatever word is being defined when **if** is executed. **compile** then advances the return address by two so that the next word in the parameter field of **if** to be executed is **forward** instead of **0branch** .

**compile** is a clear example of a word that would have to be defined differently for implementations of FORTH which do not put post-incremented return addresses on the return stack.

**copyright 1983** 

## STRING LITERALS

What does the compiler do when it runs into a string literal? For example, recall our very first FORTH word.

```
: hi ." hello" ;
```

What does the compiler do with the string so that when **hi** executes the string is printed out?

Actually, the compiler doesn't do anything with it. It turns out that ." is an immediate word, so the compiler just executes it. It is ." that has to do all of the work. What does it do? Let's look at its definition.

```
: ." ( -- ) system compile (.")
          ascii " word c@ 1+ allot ;
```

The first line compiles a special run-time word, and the second line gets the next string in the input stream delimited by the double quote mark. This word is moved to **here** but the byte at **here** contains the number of characters in the string. We use **c@** to get this count onto the stack, add one to it, and use **allot** to advance the address returned by **here** by that amount.

In short, the string, preceded by its count, is compiled into the dictionary as part of the parameter field of **hi** . Here is what **hi** looks like after it is compiled.

```
+---------------+---------------+
|           link|field          |
+---------------+---------------+
| count byte    |
+---------------+---------------+
|      h               i        |
+---------------+---------------+
|           code|field          |
+---------------+---------------+
|          cfa of (.")          |
+---------------+---------------+
|      5               h        |
+---------------+---------------+
|      e               l        |
+---------------+---------------+
|      l               o        |
+---------------+---------------+
|          cfa of exit          |
+---------------+---------------+
```

What does (.") do? It must print out the string, then it must arrange things so that exit is the next word executed after the

string is printed. This means that **(.")** must leave the IP register pointing to the right place. Here is one way of coding **(.")** .

```
: (.")  ( -- )  r> count 2dup type + >r ;
```

Notice that it uses the same trick with the return stack that **compile** used. When **(.")** starts executing, the address on the return stack is the address of the byte which holds the count of characters in the string to be printed. So, we get this address and execute **count** which adds one to the address (giving the address of the first character in the string), then pushes the count of characters in the string on top of it. These are the parameters we must give to **type** . But first, we copy the address and the count. The copies are removed by **type** then the originals are added together. Magically, the result is the address of the first byte past the string. As you can see, this address contains the execution address of **exit** so we return it to the return stack and everything works out just right.

# CHAPTER 10

# VOCABULARIES

Vocabularies are used to separate applications. For example, eFORTH is supplied with five vocabularies, **forth** , **system** , **editor** , **assembler** and **disking** . The words in these vocabularies are used in rather dissimilar situations, so they are separated in a way that allows the words they contain to be removed from dictionary searches when they aren't needed. This has the advantage of cutting down on dictionary search times during compilation. Furthermore, the same word can be defined to do different things in different vocabularies.

## CONTEXT AND CURRENT VOCABULARIES

As mentioned in Chapter 2, the context vocabulary is the vocabulary which is searched first, and the current vocabulary is the vocabulary to which new words are added. A vocabulary is made the context vocabulary by simply executing its name, and a vocabulary is made the current vocabulary by first making it the context vocabulary, then executing **definitions** .

## CREATING NEW VOCABULARIES

A new vocabulary is created with the defining word **vocabulary** followed by the name of the new vocabulary. Entering **vocabulary files immediate** will create a new vocabulary named "files". Any word which specifies a vocabulary should be declared to be an immediate word. We shall see why in a moment. Entering **files definitions** will then cause subsequently defined words to be added to the **files** vocabulary. With eFORTH, no more than ten vocabularies should be created.

## VOCABULARY CHAINING

When a vocabulary is created, it is "chained" to the current vocabulary. In eFORTH, the **system** , **editor** and **assembler** vocabularies are each chained to the **forth** vocabulary. However, the **disking** vocabulary is chained to the **system** vocabulary. This

means that if **forth** is the context vocabulary, to use words in the **disking** vocabulary, you must enter **system** then **disking** . The point of this is that the **disking** vocabulary contains very powerful words which can cause a great deal of damage to data on your disks. This scheme decreases the likelihood that they will be accidentally executed.

The vocabulary structure is a "tree" structure, and the **forth** vocabulary is the "root" of the tree. Chaining is of significance when the interpreter (or the compiler) is looking for a word in the dictionary.

## DICTIONARY SEARCHING

Whenever the dictionary is searched by words such as ' or -' or **find** , the context vocabulary is searched first. If the word is not found in the context vocabulary, the current vocabulary is searched (if it is different from the context vocabulary). If at this point the word still hasn't been found, the vocabulary to which the current vocabulary is chained is searched. This chain is followed until the **forth** vocabulary is finally reached and searched. (Vocabularies to which the context vocabulary is chained are not searched.)

Look at the definitions of **t** and **v** on block 18. We want to be able to execute them no matter what the context vocabulary is, so we put them into the **forth** vocabulary. Notice that **t** simply sets the current line then calls **v** . Let's look at **v** for a moment.

When **v** is compiled, the **forth** vocabulary is both the context and current vocabulary. If it were not also the context vocabulary, the colon would make it the context vocabulary as well. Since the definition of **v** contains a word which is in the **editor** vocabulary, it will not be found unless we do something which will result in the **editor** vocabulary being searched as well. Since the first word in its definition is **editor** , and since **editor** is an immediate word, **editor** executes, and the **editor** vocabulary becomes the context vocabulary. The current vocabulary is not changed. So, while **v** is being compiled, the **editor** vocabulary will be searched first, then the **forth** (current) vocabulary will be searched. After **v** executes we will probably want to do some editing, so when **v** executes it should make the **editor** vocabulary the context vocabulary, so the definition concludes with **[compile] editor** to cause this to happen. Here again is the important difference between what happens at compile time and what happens at run time.

Since vocabulary words are typically used inside a definition to switch the context vocabulary, it is important that vocabulary words be immediate words.

## SEALED VOCABULARIES

Remember the restaurant application we developed in Chapter 4? Once this application is being used in the restaurant, we do not want the employees to be able to execute anything other than the words defined in the application. In particular, if someone were to enter a word such as **move** the result could be disastrous. The solution is to put the application words in a separate vocabulary, then "seal" that vocabulary so that FORTH will only find words in the application. A vocabulary is sealed by "breaking" its chain.

We start by putting

```
vocabulary Menu immediate     Menu definitions
```

on line 1 of block 57. When we load block 57 all the new words will be placed into the **Menu** vocabulary. For convenience, we should add another word to the **Menu** vocabulary,

```
: ReturnToForth ( -- ) forth definitions ;
```

so that we can gracefully use FORTH again. We should also add

```
forth definitions
: RunMenu ( -- ) Menu definitions ;
```

to conveniently start the application. All that remains is to seal the **Menu** vocabulary.

```
: seal ( -- )  0 context @ 2+ c! ;
```

Now, executing **Menu seal** will do the job. When we are all done, block 57 should contain the following.

```
vocabulary Menu immediate    Menu definitions
58 load
59 load
: ReturnToForth ( -- ) forth definitions ;
forth definitions
: RunMenu ( -- ) Menu definitions ;
: seal ( -- )  0 context @ 2+ c! ;
Menu seal
ReturnToForth
```

 copyright 1983

# CHAPTER 11

## HOW CAN I "PROTECT" MYSELF?

A major difficulty FORTH newcomers have is getting used to FORTH's program control structures. A frequent mishap is to write a definition with an **if** in it and forget to put the necessary **then** after it. Sometimes different structures are incorrectly combined. For example,

**... do ... if ... then ... loop**

is just fine but

**... do ... if ... loop ... then**

is definitely not "ok" with FORTH. The program structuring words in the pre-compiled portion of eFORTH do not check for any of these "mistakes". They assume that any words being compiled are correct, so they do not waste any time performing this kind of checking. Indeed, the eFORTH electives and extensions have been thoroughly tested, so the only thing FORTH has to do is compile them into the dictionary as rapidly as possible. We should not have to extend our wait while redundant and unnecessary "error" checking is going on.

### COMPILER SECURITY

However, when you are developing an application, it would be nice if FORTH did this kind of checking to prevent you from wasting time trying to find what went wrong. Blocks 39 and 40 contain redefinitions of the program structuring words. These new versions perform a simple syntax check and print error messages if something isn't right. This is referred to as "compiler security". A **do** must be correctly terminated by a **loop** or **+loop** or else!

Block 41 contains redefinitions of the colon and semicolon. The new definition of the colon makes sure you don't leave off the semicolon of the previous word. The new definition of the semicolon makes sure that the definition did not change the stack. Presumably, changing the stack means that you used an if without a **then** or committed some similar crime. Block 41 also redefines the word which is executed by **create** so that if you

 copyright 1983

redefine a word you will be told about it. Unintentionally redefining a word can lead to a great deal of head scratching.

## DISK ERRORS

Block 3 contains a redefinition of **(r/w)** which is the disk access word executed by **r/w** because **r/w** does not check for or report disk errors, but it does save the status code returned by the last disk access. The new version of **(r/w)** checks this status and reports any error and aborts. This gives you the flexibility to check for disk errors in your applications and recode **(r/w)** to perform whatever operation you feel is appropriate when a disk error occurs. Again, FORTH does not take control away from you. It gives you the power (and responsibility) to decide what to do when exceptional conditions occur.

## EXECUTION VARIABLES

The ability to redefine the behavior of low-level FORTH operations is based upon the very powerful but dangerous device called an "execution variable". For example, **r/w** is a very simple word.

```
: r/w   system 'r/w @ execute ;
```

It simply gets an execution address out of the system variable **'r/w** and executes it. So, to change the behavior of **r/w** all we have to do is define a word and put its execution address into **'r/w** . This is what is done on Block 3. Notice that the new version of **(r/w)** first executes the old version, then checks for errors. Notice also that the word **protect** is executed once the new version of **(r/w)** is installed in **'r/w** . The reason for this is quite simple. Once installed, we do not want the new version to be removed from the dictionary with **forget** or **empty** because the word executed by **r/w** would no longer exist. **protect** changes the system so that everything in the dictionary when it executes will stay there as long as FORTH is running.

# CHAPTER 12

## THE eFORTH 6809 ASSEMBLER VOCABULARY

The assembler vocabulary is used when you need operations that have not yet been implemented in FORTH (such as processing interrupts and other hardware capabilities) or when a process needs to be as fast possible. And it is the ability to code some words in machine language that makes FORTH an ideal programming tool in environments where one must have complete control of a computer's hardware and peripherals. This section assumes you are familiar with 6809 assembly language.

The assembler vocabulary is invoked automatically when the words **code** and **;code** are used.

### code DEFINITIONS

**code** is used to create a word whose behavior will be specified with assembly language mnemonics instead of high-level FORTH code. For example, here is the definition of + for the 6809 using the eFORTH assembler.

```
code  +    ( n1 n2 -- n3 )
            d pulu   0 ,u addd   0 ,u std
            next   end-code
```

The 6809 U register is used for the FORTH machine's SP register. So when + is executed, **d pulu** pulls the 16-bit number on top of the stack and puts it into the D accumulator. Next, the number now on top of the stack is added to the D accumulator by **0 ,u addd** and replaced with the result by **0 ,u std.** Finally, **next** is a macro which compiles the 6809 code for NEXT.

You should recall that NEXT is the routine which is executed between each FORTH word, and every word in FORTH must ultimately jump to NEXT.

The word **end-code** is used to signal the end of a **code** or **;code** definition. It restores the context vocabulary to what it was before the assembler vocabulary was called.

 copyright 1983

As you may have noticed, even assembly code is written in reverse order in FORTH. Basically, the rule is that all operands must be specified before writing the mnemonic.

Here is what happens when the interpreter sees a **code** definition. When **code** is executed it creates the name field, the link field, and sets the code field to point to the parameter field. The mnemonics which follow the name put the appropriate machine codes in the parameter field.

It is important to point out that, unlike the colon, **code** does not put FORTH into the compiling state. All the words which follow it are executed. This means that each mnemonic must be defined so that when it executes it compiles the proper machine code for that mnemonic into the dictionary. No words in the assembler vocabulary are immediate. Consequently, it is an easy matter to write macros. For example, by defining next as

```
: next   ,y++ ldx   0 ,x ] jmp ;
```

it becomes a macro-instruction to compile code for several machine instructions. Obviously, such macros can be defined to use parameters passed to them on the stack.

Let's look at the definitions of **2@** and **2!** . Double precision variables must have four bytes reserved in their parameter fields. We shall specify that the byte in the parameter field with the lowest address holds the most significant byte of the 32-bit variable. So, to push the four bytes in the parameter field of a 32-bit variable we would code it as follows:

```
code  2@    ( adr -- d )
    x pulu   2 ,x ldd   d pshu   0 ,x ldd   d pshu
    next     end-code
```

Since the 6809 U register serves as the FORTH machine's SP register, we pull the address on top of the FORTH stack into the 6809 X register, then load the D accumulator with the low 16 bits of the 32-bit variable and push them to the stack. The next line of code loads the D accumulator with the high 16 bits and pushes them to the stack.

Similarily, **2!** could be coded as

```
code  2!     ( d adr -- )
        x pulu   d pulu   ,x++ std   d pulu   0 ,x std
        next     end-code
```

The address of the variable is pulled into the X register and the high 16 bits are pulled into the D accumulator. These are stored

at the address and the X register is incremented twice to point to the low 16 bits in the variable. The low 16 bits are pulled from the stack and stored in the variable. Finally, NEXT is executed.

### ;code DEFINITIONS

As you might suspect, defining a word that defines other words is a bit more complicated. Let us define **2constant** which when executed will add 32-bit constants to the dictionary. As with **constant** , we shall suppose that the constant to be entered into the dictionary is on the stack when **2constant** is executed.

```
: 2constant   ( d -- )
     constant ,   ;code
     4 ,x ldd   d pshu   2 ,x ldd   d pshu
     next   end-code
```

When **2constant** is executed, it executes **constant** which creates a dictionary entry and sets the code field to point to the routine which pushes 16-bit constants to the stack and puts the 16-bit number on top of the stack into the parameter field. The comma puts the next 16-bit stack item into the dictionary which means that the parameter field being created now contains the 32-bit constant. But the code field, you recall, points to the routine which pushes 16-bit constants to the stack. This is remedied by **;code** which overwrites the code field so that it points to the code which follows **;code** .

So, when you enter

**10000. 2constant sample**

**sample** will be added to the dictionary. Its parameter field will contain the 32-bit representation of 10,000, and its code field will point to the machine code which follows **;code** in the definition of **2constant** .

This code gets the contents of W which points to the code field of the double precision constant being executed. In eFORTH's 6809 implementation of the FORTH machine, the W register is implemented with the X register, so on entry to the machine code we may assume that X is pointing to the code field address of the double constant being executed.

Actually, we could have defined **2constant** a bit more economically. This definition illustrates an important feature of the assembler.

```
: 2constant    ( d -- )
      constant,    ;code
      2 ,x leax   ' 2@ 2+ 2+ jmp   end-code
```

Among other things, **;code** stops compilation which means all the words which follow it are executed rather than compiled into the dictionary. This allows the programmer to use high-level FORTH to calculate operands which is what is done in the above definition. First, the **leax** instruction is compiled. Next, the phrase **' 2@** pushes the code field address of **2@** to the stack. Then we add two to it to get the parameter field address. Now, if we look at the code for **2@** , the instructions beginning with **2 ,x ldd** on the second line are exactly what we want a double precision constant to do. This instruction is located in the second byte of the parameter field of **2@** so, we add two to the parameter field address of **2@** ( which is on the stack) and use this as the operand for a jump instruction.

## BRANCH INSTRUCTIONS AND PROGRAM STRUCTURE

Mnemonics for conditional branch instructions are not included. Instead, the following control structures are provided in the eFORTH assembler. They automatically compile the appropriate branch instructions to implement the structure.

```
<condition> if...then
<condition> if...else...then
begin...<condition> until
begin...<condition> while...repeat
begin...again
```

These words may look identical to the same control words available in high-level FORTH but they are quite different. This is a clear example of how the same words can be defined differently in different vocabularies.

**if** , **while** and **until** must be preceded by a condition code. The available condition codes are

```
eq  mi  hi  ls  cc  cs  vc  vs  pl
mi  ge  lt  gt  le  lo  hs
```

and the condition specified by any of them may be inverted with **not** .

The phrase **eq if** will cause the "true" part of a conditional to be executed if the z-bit of the condition code register is set; otherwise control will branch to the code which follows the subsequent **else** or **then** . Similarly, **mi until** will cause the

loop to be terminated if the n-bit of the condition code register is set according to the rule for a BMI instruction branch. Otherwise control branches to the previous **begin** .

The sequence **eq not while** will cause the code following **while** to be executed if the z-bit of the condition code register is clear; otherwise execution continues with the code following the subsequent **repeat** .

The other condition codes are the 6809 conditional branch mnemonics without the 'B'. So, the phrase **hi if** will cause the "true" part to be executed if the condition code register satisfies the conditions which would cause a BHI instruction to branch.

The words **if** , **while** and **else** all compile a (short) relative branch instruction into the dictionary, so it is possible to get a "relative branch too long" error message if, for example, you put an enormous amount of code between an **if** and its corresponding **else** or **then** , or a **begin** and its corresponding **until** or a **while** and its corresponding **repeat** . This condition is not detected until the forward branches of these words are resolved. (See their definitions on block 8.)

For straight-forward examples see the definition of **roll** on block 11 and the definition of **du<** on block 26. For a very non-straight-forward example, see the definition of **-match** on block 19. The stack is manipulated with **swap** and **rot** to move around the addresses marking forward references which need to be resolved. The result is very unstructured but byte efficient code.

## eFORTH ASSEMBLER SYNTAX

The mnemonics provided in the eFORTH assembler vocabulary are listed here according to the number and type of operands they require. The eFORTH syntax follows Motorola's "green card" except, as noted earlier, operands are given before the mnemonic. The syntactic symbol <number> is used to represent any sequence of FORTH words which leave a 16-bit number on the stack. The symbol <mmm> is used to represent an arbitrary mnemonic.

## IMMEDIATE ADDRESSING

The immediate addressing mode is specified by preceding the mnemonic with the usual "#" sign.

```
<number> # <[illegible]>
```

The following code would be used, for example, to compare the contents of the A accumulator to the ASCII carriage return code:

```
13 # cmpa
```

## EXTENDED ADDRESSING

The extended addressing mode is specified for the 6809 by simply preceding the mnemonic with the address. Extended addressing is the default addressing mode unless immediate, direct, or indexed addressing is explicitly specified.

```
<number>  <[illegible]>
```

## DIRECT ADDRESSING

In the direct addressing mode the byte following the mnemonic is combined with the 6809 direct page register to form an effective address. Direct addressing must be explicitly specified for the 6809 with the symbol "<" placed preceding the mnemonic. The eFORTH 6809 implementation uses the direct page register as a pointer to the user variable area. Consequently, direct addressing will access the user variables of the current user.

```
buffer  <  ldx
```

## INDEXED ADDRESSING

The constant-offset indexed addressing mode is specified by preceding the mnemonic with the name of an indexable register preceded with a comma. This, in turn, must be preceded with a number which specifies the constant offset. Note that a constant offset of zero must be explicitly given. Hence a constant offset from the U register is specified with

```
2 ,u ldd
```

In addition, the program counter can be used with a constant offset. For example,

```
table ,pcr leax
```

The accumulator offset indexed addressing mode is specified by using one of the following:

```
a,x   b,x   d,x     a,y   b,y   d,y
a,u   b,u   d,u     a,s   b,s   d,s
```

For example,

```
b,y lda
```

is equivalent to the standard 6809 assembly code

```
LDA  B,Y
```

The auto increment or auto decrement addressing mode is specified by preceding the mnemonic with one of the following words:

```
,x+   ,y+   ,u+   ,s+   ,x++   ,y++   ,u++   ,s++
,-x   ,-y   ,-u   ,-s   ,--x   ,--y   ,--u   ,--s
```

The indirect addressing mode is specified by preceding the mnemonic with a square bracket. For example,

```
<number> ] ldx
```

and notice that the bracket must be separated on both sides with spaces. If the words inside the bracket just push a number to the stack, as in the previous example, the addressing mode will be extended indirect. The bracket may also follow words which specify other addressing modes to give the indirect version of that mode. Constant offset indexed indirect addressing is specified with

```
0 ,u ] ldd
```

Accumulator offset indexed indirect addressing is specified with

```
b,y ] lda
```

Auto double increment indexed indirect addressing is specified

with

```
,x++ ] ldd
```

Program counter constant offset indirect addressing is specified with

```
table ,pcr ] leax
```

## RELATIVE ADDRESSING

Relative addressing is only used by two eFORTH assembler mnemonics:

```
<number> bra
<number> bsr
```

and <number> is taken to be the absolute address to branch to. The assembler will generate an 8-bit or 16-bit operand as required.

## 6809 MNEMONICS

The bulk of the 6809 opcodes can be divided into three classes: (1) those which are used without any operands, (2) those which must be used with either direct, extended, or indexed addressing modes but which cannot be used with the immediate addressing mode, and (3) those which may be used with the immediate addressing mode or with one of the other three major addressing modes.

## MNEMONICS - NO OPERANDS

The following mnemonics are used alone. They neither require nor use any operands placed on the stack.

```
nop   sync  daa   sex   abx   mul
rts   rti   swi   swi2  swi3
nega  coma  lsra  rora  asra  asla  rola  deca  inca  tsta  clra
negb  comb  lsrb  rorb  asrb  aslb  rolb  decb  incb  tstb  clrb
```

## MNEMONICS - IMMEDIATE ADDRESSING ILLEGAL

The mnemonics listed here require an operand which specifies either direct, extended, or indexed addressing mode. The immediate addressing mode is illegal, but no error message will be given.

**neg com lsr ror asr asl**
**rol dec inc tst clr**
**sta stb std stx sty stu sts**
**jsr jmp**

## MNEMONICS - IMMEDIATE ADDRESSING PERMITTED

The mnemonics listed here must be preceded with words which specify immediate, direct, extended, or indexed addresssing modes.

**suba subb subd**
**adda addb addd**
**cmpa cmpb cmpd cmpx cmpy cmpu cmps**
**lda ldb ldd ldx ldy ldu lds**
**sbca anda bita eora adca ora**
**sbcb andb bitb eorb adcb orb**

## MNEMONICS - IMMEDIATE OPERANDS REQUIRED

These mnemonics assume immediate addressing and use the number on the stack for the immediate operand

**andcc orcc cwai**

## MNEMONICS - INDEXED ADDRESSING REQUIRED

The following mnemonics must be preceded with words which specify one of the indexed addressing modes. This includes the program counter constant offset mode and the indirect indexed modes.

**leax leay leau leas**

 copyright 1983

## MNEMONICS - REGISTER OPERANDS REQUIRED

The mnemonics listed here may only have one or more registers specified as operands.

```
puls   pulu   pshs   pshu   tfr   exg
```

For example,

```
a b dpr x y  pulu
```

is the code to pull the A and B accumulators, the direct page register, and the X and Y registers from the stack pointed to by the U register. And

```
a dpr tfr
```

will transfer the contents of the A accumulator to the direct page register. The legal register names are

```
a  b  d  x  y  u  s  pcr   dpr   ccr
```

## MACROS

Words can be defined in terms of the available mnemonics to produce macros or define new mnemonics. For example, an ASLD mnemonic could be added to the 6809 repoitoire with

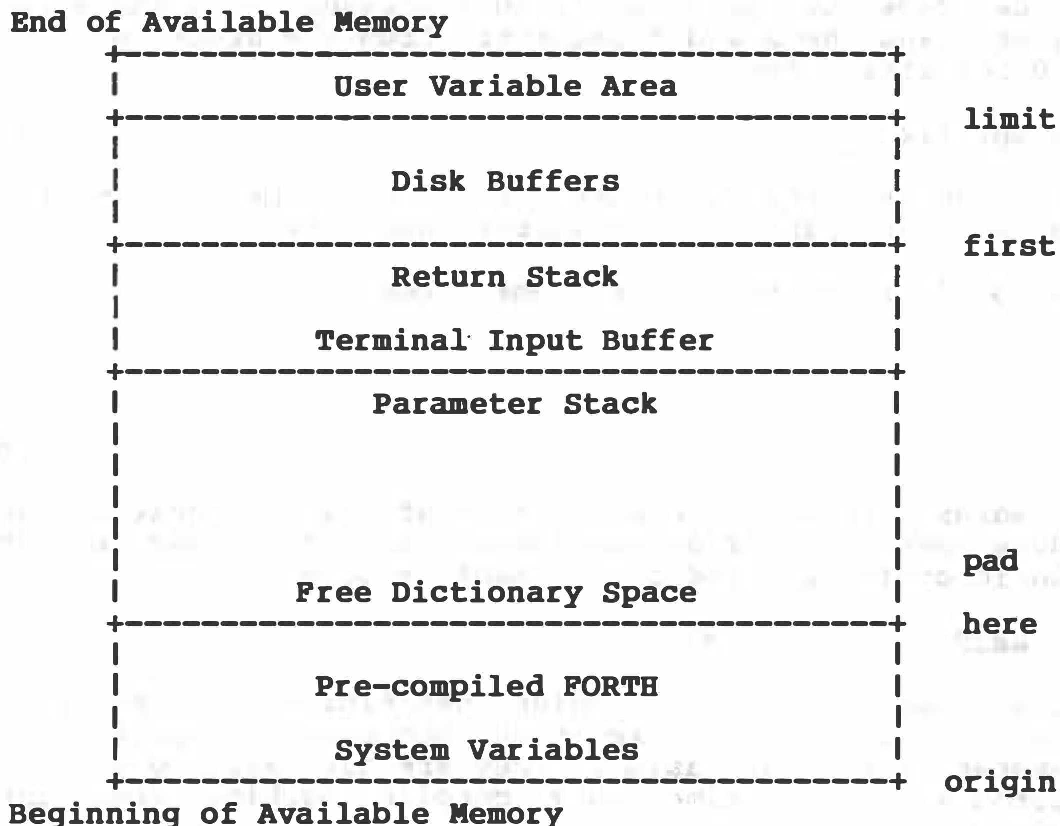
```
: asld   aslb   rola ;
```

Notice that this is a colon definition so the **asl** and rol mnemonics have their execution addresses compiled into the parameter field of **asld** . They are not executed until **asld** is executed at which time they compile machine code into the dictionary.

# CHAPTER 13

## WHERE DOES eFORTH PUT THINGS?

eFORTH uses memory in accordance with this memory map.




### THE DICTIONARY

The dictionary starts in low memory and grows upward as words are defined. The word **here** returns the address of the first free byte in the dictionary. Words can be removed from the dictionary with **forget** or **empty** , and memory released by this process is reclaimed.

## THE PARAMETER STACK

The starting address of this stack is contained in the variable **s0** , and **'s** returns the address of the last number pushed to the stack. The stack grows downward toward the dictionary. It is possible for the stack and the dictionary to collide. eFORTH does not check for this condition.

## THE TERMINAL INPUT BUFFER

This buffer is reserved to hold a line of text entered from the keyboard. Characters are stored here beginning at the address contained in the variable **s0** moving upward toward the return stack.

## THE RETURN STACK

This stack is used to hold return addresses and various sorts of temporary data. Its origin is contained in the variable **r0** , and the word **'r** returns the address of the last number pushed to this stack. This stack grows downward toward the terminal input buffer. They share 256 bytes which is more than adequate.

## THE DISK BUFFERS

eFORTH reserves 1028 bytes for each disk buffer (1024 are used to hold the data on a block) and reserves space for four buffers when it starts running.

## THE USER VARIABLE AREA

The address at which this area begins is returned by **'u** . This area contains user variables and allows eFORTH to be expanded for multi-programming.

# CHAPTER 14

# THE END OF THE TOUR

This concludes our tour of FORTH and some of the intimate details of eFORTH. I have found FORTH to be an ideal programming environment. It doesn't force things on me, and it allows me to interactively explore my hardware and develop high-level applications. Despite the fact that I know dozens of programming languages and teach in a computer science department where Pascal is the major instructional language (soon to be replaced by Modula-2), whenever I have a choice, I choose FORTH. I have written a multi-tasking system that allows me to start any number of programs running, all of which can communicate with one another, turtle graphics, music synthesis, and a variety of file and data-base structures. I hope that FORTH helps you to be as productive as much as it has helped me.

## LITERAL STRINGS

A few odds and ends haven't been discussed that I would like to mention before leaving you. eFORTH gives you the ability to use literal strings. The word " ("quote"), which is defined on block 31, is used in a number of places. If you study them, you should have no trouble using it. It is used in the definition of **date** on block 66 and in **header** on block 31. Notice that **." hello"** is equivalent to **" hello" type** (except that quote is a "smart" word but dot-quote isn't). Quote is immediate, and whenever it executes, it puts the address and count of the string which follows it (to the terminating quote) onto the stack. However, if FORTH is in the compilation state, it will compile a run-time word, then the string into the parameter field of the word being defined. Later, when the word being defined executes, the run-time word compiled by quote will push the address and count of the string to the stack.

It is used in **header** to just print out the string, but in **date** a substring is extracted from the string.

## SMART WORDS

The quote is a smart word; it behaves one way inside a definition (it compiles), and another way outside of a definition (it moves the string to **pad** ). In general, smart words are being discouraged these days, but quote strikes me as being a rather benign one. The word **ascii** , defined on block 13, is also a smart word.

## A CASE STRUCTURE

When Chapter 11 called your attention to block 38 you may have wondered how to use those words. (Notice that block 40 contains "secure" versions. Versions without compiler security are defined on block 38.) Here are two samples.

```
   : is ( n -- )
       case
         1 <of   ." Less than one."        else
         1  of   ." One."                  else
       2 5 range ." Two-Five."             else
         5 >of   ." Greater than five."    else
       endcase   ;
```

Test this word by entering things like **4 is** and hitting return.

```
   : equals ( adr cnt -- )
       case
         " one" "of 1 .     else
         " two" "of 2 .     else
         " ten" "of 10 .    else
         ." What?"
       endcase   ;
```

This last one is tested by entering things like

```
   " one" equals
   " three" equals
   " ten" equals
```

but be ready for a surprise when you try **" tenth" equals .** Oh, well, nobody's perfect.

## APPENDIX A

### HOW DOES eFORTH DIFFER FROM "Starting FORTH"?

eFORTH was designed to follow contemporary FORTH standards. The original intention was to follow the FORTH-83 STANDARD, however, at this writing, the standard hasn't been published. Accordingly, eFORTH follows the FORTH described in Brode's **Starting FORTH** except in those cases where we are fairly sure what will be in FORTH-83.

### LOOPS

Perhaps the most significant difference is in the behavior of the **do...loop** structure. The behavior of the eFORTH implementation is described in Chapter 6. Other differences which should be mentioned are these.

First, the word **i** does not simply return the number on top of the return stack. It must perform a calculation on it. There are situations where Brode does not use it inside a loop (pp. 111-112). This will not work in eFORTH. The words **i** and **j** must only be used inside a loop, and only to return the current loop index. To move a copy of the number on the return stack to the parameter stack, you must use **r@** in eFORTH. There is no word in eFORTH which is equivalent to **I'** in Brode.

The word **DOUBLING** defined on page 134 is not restricted to an upper limit of 32,767 in eFORTH. Try 65,525, or try zero. The word **TEST** defined on page 135 will behave quite differently in eFORTH. The eFORTH loop implementation eliminates the need for **/LOOP** described by Brode on page 162.

### execute

In **Starting FORTH** , the word **execute** expects a word's parameter field address on the stack. In eFORTH, **execute** expects a word's code field address (execution address). This is also true of ' ("tick") which returns a code field address in eFORTH, but a parameter field address in **Starting FORTH** . However, all of the examples which use them in **Starting FORTH** will also work in eFORTH.

**cmove AND <cmove**

These words generally behave in the manner described on page 267 except that, when possible, they will move two bytes at a time.

**?stack**

This word does not return a flag. In eFORTH it will abort if there has been stack underflow. This follows the consistent naming convention that words whose names begin with a question mark contain some sort of conditional execution which may result in an abort. If a word simply returns a flag, the question mark should be at the end of its name.

**NUMBER FORMATTING**

Use the "set-up" phrases in the box on the top of page 172.

## APPENDIX B

## eFORTH MASTER GLOSSARY

This glossary contains an entry for each word supplied with eFORTH except for those which are implementation specific. Words which are supplied only for a particular implementation are described in the appendix which describes that implementation.

These entries are listed according to their ASCII order. The first line gives the name of the word being described, the vocabulary in which it is found, the block number from which it was loaded (a zero means that it was not loaded from a block), and its stack effect. The remaining lines give a brief description of what the word does.

In the stack effect, the two dashes indicate the point at which the word executes. The parameters which must be placed on the stack before the word is executed are on the left; the values the word returns are on the right. In both cases, the item on top of the stack is on the right.

The symbols used to indicate stack items include:

| | |
|---|---|
| b | 8-bit byte (the high 8-bits are zero) |
| c | 7-bit ASCII character (the high 9-bits are zero) |
| n | 15-bit signed integer |
| u | 16-bit unsigned integer |
| d | 31-bit signed integer |
| ud | 32-bit unsigned integer |
| flg | boolean flag (zero is false, non-zero is true) |
| tf | true boolean flag (non-zero) |
| ff | false boolean flag (zero) |
| adr | 16-bit memory address |

The sequence "adr cnt" is frequently used and referred to as specifying a string. Specifically, "adr" represents the address of the first character in the string, and "cnt" represents the number of characters in the string.

copyright 1983 

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**!** **forth** **0** **( n adr -- )**
Store n at adr.

**"** **forth** **13** **( -- )**
Compile a literal string with a run-time word which will push its address and count to the stack. If not compiling, move the word to pad and push its address and count to the stack.

**"of** **forth** **38** **( -- adr )**
Begins a phrase to be executed if the case select string is equal to the string identified on the stack; otherwise execution branches to the words which follow the next "else". See ("of) .

**#** **assembler** **0** **( -- )**
Specify the immediate addressing mode.

**#** **forth** **0** **( ud1 -- ud2 )**
Generate from an unsigned double number, ud1, the next ASCII character which is placed in the output string. Result ud2 is the quotient after division by base and is held for further processing.

**##** **forth** **10** **( b -- )**
Print b as two hex digits.

**####** **forth** **10** **( u -- )**
Print u as four hex digits.

**#>** **forth** **0** **( d -- adr cnt )**
Terminate pictured numeric output conversion. Leave the address and count of the string. May be followed by type.

**#f** **editor** **20** **( -- adr )**
Return the address of the editor's find buffer.

**#i** **editor** **20** **( -- adr )**
Return the address of the editor's insert buffer.

**#s** **forth** **0** **( ud -- 0 0 )**
Convert all digits of an unsigned 32-bit number adding each to the output string until the remainder is 0. At least one digit is generated. Use between <# and #>.

**'** **forth** **0** **( -- cfa )**
Search the dictionary for the next word in the input stream. Leave its execution address if found. Abort if it isn't found.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**'bell** system 7 ( -- adr )
A system variable which contains the execution address of the word executed by bell. Its initial value is (bell).

**'bs** system 7 ( -- adr )
A system variable which contains the execution address of the word executed by bs. Its initial value is (bs).

**'claim** disking 50 ( -- adr )
Return the address of a word executed by Claim which performs system dependent functions.

**'config** disking 50 ( -- adr )
Return the address of the word executed by Configure.

**'cr** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by cr for the current output device.

**'create** system 7 ( -- adr )
A system variable which contains the execution address of the word executed by create. Its initial value is (create).

**'depth** system 14 ( -- adr )
Return the address which holds the maximum number of lines on the current output device.

**'device** system 14 ( b -- )
Create a name for the field which is offset b bytes from the beginning of the parameter field of a device word. Standard device words are term and printer.

**'emit** system 7 ( -- adr )
Returns the address of the system variable which holds the execution address of the word executed by emit. Its initial value is (emit).

**'eol** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by eol for the current output device.

**'eos** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by eos for the current output device.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**'expect** system 7 ( -- adr )
A system variable which contains the execution address of the word executed by expect. Its initial value is (expect).

**'get** system 14 ( -- adr )
A user variable which holds the address of the parameters for the current input device.

**'home** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by home for the current output device.

**'key** system 7 ( -- adr )
Return the address of the system variable which holds the execution address of the word executed by key. Its initial value is (key).

**'key?** system 7 ( -- adr )
A system variable which contains the execution address of the word executed by key?. Its initial value is (key?).

**'number** system 7 ( -- adr )
Return the address of the system variable which holds the execution address of the word which does input number conversion. Its initial value is (number), but is usually set to number.

**'page** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by page for the current output device.

**'put** system 14 ( -- adr )
A user variable which holds the address of the parameters for the current output device.

**'r** forth 0 ( -- adr )
Return the contents of the return stack pointer.

**'r/w** system 7 ( -- adr )
Return the address of the system variable which holds the execution address of the word executed by r/w. Its initial value is (r/w).

**'s** forth 0 ( -- adr )
Return the contents of the parameter stack pointer.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**'start** system 7 ( -- adr )
Return the address of the system variable which holds the first FORTH word to be executed on a cold start. Its initial value is quit.

**'type** system 7 ( -- adr )
Return the address of the system variable which holds the execution address of the word executed by type. Its initial value is (type).

**'u** forth 0 ( -- adr )
Return the base address of the active user variable area.

**'update** editor 18 ( -- adr )
An execution variable which holds the execution address of the word to be executed whenever changes are made to the current editing block.

**'width** system 14 ( -- adr )
Return the address of the line width value for the current output device.

**'xy** system 14 ( -- adr )
Return the address which holds the execution address of the word executed by xy for the current output device.

**(** forth 0 ( -- )
Forces the interpreter to skip any text between this word and the next ')'.

**(")** system 13 ( -- adr cnt )
Run-time word compiled by " which returns the address and count of the literal string which was between the quotes.

**("of)** system 37 ( a1 c1 a2 c2 -- a1 c1 )
Run-time word compiled by "of . All four values are dropped if the strings are identical; otherwise a1 and c1 are left and execution branches to the next case.

**(+loop)** system 0 ( n -- )
Similar to (loop) except that n is added to the index. If this results in crossing the boundary between the index and the index minus one, the loop is terminated.

copyright 1983 

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**(.")** system 0 ( -- )
Run-time word compiled by ." .

**(;code)** system 0 ( -- )
The run-time word compiled by ;code .

**(<of)** system 37 ( n1 n2 -- n1 )
Run-time word compiled by <of . Both n1 and n2 are dropped if n1 is less than n2; otherwise n1 is left and execution branches to the next case.

**(>of)** system 37 ( n1 n2 -- n1 )
Run-time word compiled by >of . Both n1 and n2 are dropped if n1 is greater than n2; otherwise n1 is left and execution branches to the next case.

**(?do)** system 0 ( limit index -- )
Run-time word compiled by ?do. index is the initial index and limit is the loop limit. If limit is less than or equal to index, the loop is not executed.

**(?leave)** system 0 ( flg -- )
A run-time word which forces immediate termination of the currently executing loop if the flag is non-zero.

**(abort)** system 0 ( flg -- )
The run-time word compiled by abort" . If flg is non-zero, the in-line text which follows is printed and quit executed, otherwise execution branches to the first word which follows the text.

**(bell)** system 0 ( -- )
Sound the "bell" on the current output device.

**(bs)** system 0 ( -- )
Transmit a destructive backspace to the current output device.

**(cr)** system 33 ( -- )
Issue a carriage return and line feed to the current output device.

**(create)** system 4 ( -- )
Used in the form create www to create a dictionary entry for www. When www executes, it will return the address of it's parameter field unless subsequently modified by does> or ;code .

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**(do)** **system** **0** **( limit index -- )**
Run-time word compiled by do. index is the initial index and limit is the loop limit. If limit equals index, the loop is executed 64K times (if terminated by loop).

**(emit)** **system** **0** **( c -- )**
Transmit the ASCII coded character on the stack to the current output device.

**(expect)** **system** **0** **( adr cnt -- )**
Accept a maximum of cnt characters from the current input device storing them at adr. Input is terminated when a carriage return is received.

**(forget)** **system** **0** **( cfa -- flg )**
Forget the word whose execution address is given (and forget all words since it was defined). Leave a zero if the operation was successful; leave a non-zero if the operation was aborted.

**(key)** **system** **0** **( -- c )**
Wait for a character to be received from the current input device, then push its ASCII code to the stack.

**(key?)** **system** **0** **( -- flg )**
Return a true flag if a key has been pressed on the terminal; otherwise return a false flag.

**(leave)** **system** **0** **( -- )**
A run-time word which forces immediate termination of the currently executing loop. See leave.

**(literal)** **system** **0** **( -- n )**
The run-time word compiled by literal . When executed, the 16 bits which follow it are pushed to the stack.

**(loop)** **system** **0** **( -- )**
The run-time word compiled by loop . When executed, the loop index on the return stack is incremented and the loop is terminated if the index equals or exceeds the loop limit; otherwise, execution branches to the previous do .

**(number)** **system** **0** **( adr -- n )**
Convert the string whose count byte is at the specified address using the current base. A single precision number is returned. Aborts if conversion is not possible. The byte at adr is not used.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**(of)** **system** **37** **( n1 n2 -- n1 )**
Run-time word compiled by of . Both n1 and n2 are dropped if they are equal; otherwise n1 is left on the stack and execution branches to the next case.

**(r/w)** **system** **0** **( adr blk flg -- adr )**
If the flag is non-zero, the specified block is read from disk and stored in memory beginning at the specified address; otherwise, 1024 bytes beginning at the specified address are written to the specified block on the disk.

**(range)** **system** **37** **( n1 lo hi -- n1 )**
Run-time word compiled by "range". All three numbers are dropped if n1 is "within" lo and hi; otherwise n1 is left and execution branches to the next case.

**(type)** **system** **0** **( adr cnt -- )**
Transmit cnt characters beginning at adr to the current output device.

***** **forth** **0** **( n1 n2 -- n3 )**
Signed multiply of n1 by n2 leaving a 16-bit result.

***/** **forth** **27** **( n1 n2 n3 -- n4 )**
Multiply n1 by n2 leaving a 32-bit result which is divided by n3.

***/mod** **forth** **0** **( u1 u2 u3 -- u4 u5 )**
Multiply u1 by u2 leaving a 32-bit intermediate result, then divide by u3 giving remainder u4 and quotient u5. All values are unsigned.

**+** **forth** **0** **( n1 n2 -- n3 )**
Return the signed sum of n1 with n2.

**+!** **forth** **0** **( n adr -- )**
Add n to the 16-bit value at adr.

**+load** **forth** **0** **( n -- )**
Begin interpretation of the block which is n blocks away from the block on which +load appears. When finished, interpretation continues with the words following +load.

**+loop** **forth** **40** **( adr1 adr2 -- )**
Use only in a definition. Marks the end of a definite loop structure. See (+loop).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| **,** | **forth** | **0** | **( n -- )** |

Allot two bytes of dictionary space and store the number on top of the stack into them.

| | | | |
|---|---|---|---|
| **,--s** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of two-byte auto-decrement on the S register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,--u** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of two-byte auto-decrement on the U register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,--x** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of two-byte auto-decrement on the X register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,--y** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of two-byte auto-decrement on the Y register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,-s** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of one-byte auto-decrement on the S register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,-u** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of one-byte auto-decrement on the U register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,-x** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of one-byte auto-decrement on the X register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,-y** | **assembler** | **0** | **( -- post )** |

Specify the addressing mode of one-byte auto-decrement on the Y register. The appropriate post byte is left on the stack.

| | | | |
|---|---|---|---|
| **,pcr** | **assembler** | **0** | **( adr -- ??? )** |

Specify the program counter relative addressing mode. adr is the absolute address of the operand. Stack effect varies depending on the distance to adr.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**,s** **assembler** **0** **( n -- ??? )**
Specify the addressing mode as a constant offset from the S register. Stack effect varies depending on the size of n.

**,s+** **assembler** **0** **( -- post )**
Specify the addressing mode of one-byte auto-increment on the S register. The appropriate post byte is left on the stack.

**,s++** **assembler** **0** **( -- post )**
Specify the addressing mode of two-byte auto-increment on the S register. The appropriate post byte is left on the stack.

**,u** **assembler** **0** **( n -- ??? )**
Specify the addressing mode as a constant offset from the U register. Stack effect varies depending on the size of n.

**,u+** **assembler** **0** **( -- post )**
Specify the addressing mode of one-byte auto-increment on the U register. The appropriate post byte is left on the stack.

**,u++** **assembler** **0** **( -- post )**
Specify the addressing mode of two-byte auto-increment on the U register. The appropriate post byte is left on the stack.

**,x** **assembler** **0** **( n -- ??? )**
Specify the addressing mode as a constant offset from the X register. Stack effect varies depending on the size of n.

**,x+** **assembler** **0** **( -- post )**
Specify the addressing mode of one-byte auto-increment on the X register. The appropriate post byte is left on the stack.

**,x++** **assembler** **0** **( -- post )**
Specify the addressing mode of two-byte auto-increment on the X register. The appropriate post byte is left on the stack.

**,y** **assembler** **0** **( n -- ??? )**
Specify the addressing mode as a constant offset from the Y register. Stack effect varies depending on the size of n.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**,y+** **assembler** **0** **( -- post )**
Specify the addressing mode of one-byte auto-increment on the Y register. The appropriate post byte is left on the stack.

**,y++** **assembler** **0** **( -- post )**
Specify the addressing mode of two-byte auto-increment on the Y register. The appropriate post byte is left on the stack.

**-** **forth** **0** **( n1 n2 -- n3 )**
Return the signed result of subtracting n2 from n1.

**-'** **forth** **0** **( -- adr flg )**
Search the dictionary for the next word in the input stream. If found, return a false flag and the execution address of the word; otherwise leave a non-zero flag and here .

**-->** **forth** **0** **( -- )**
Stop interpretation of the current block and continue interpretation with the next sequential block. May be used within a colon definition that crosses a block boundary.

**-match** **editor** **19** **( A U a u -- a1 flg )**
Search for the string at a in the string at A. If found, return a false flag and set a1 to point to the character which follows the string. Otherwise return a true flag and set a1 equal to A+U.

**-search** **editor** **21** **( -- flg )**
Starting at the current cursor position search for the string in the find buffer. Limit the search to the current editing block. Returns a false flag if the string is found.

**-text** **forth** **12** **( a1 u1 a2 -- flg )**
Compare two strings. Return a false flag if they are equal; a positive number if the string at a1 is "greater" than the string at a2; a negative number if the string at a1 is "less" than the string at a2.

**-trailing** **forth** **0** **( adr u1 -- adr u2 )**
Adjust the character count u1 to exclude trailing blanks.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**.** forth 0 ( n -- )
Print n followed by one space.

**."** forth 0 ( -- )
Use only in a definition. When the word being defined is executed the text between the quotes will be printed.

**.(** forth 0 ( -- )
Immediately print the text which follows until the first right parenthesis.

**.r** forth 0 ( n u -- )
Print n right adjusted in a field u characters wide.

**.s** forth 10 ( -- )
Print the current values on the computation stack. This operation does not modify the stack in any way.

**/** forth 0 ( nl n2 -- quo )
Return the signed result of dividing nl by n2.

**/mod** forth 0 ( ul u2 -- u3 u4 )
Unsigned divide of ul by u2 leaving unsigned remainder u3 and quotient u4.

**0.** forth 25 ( -- d )
Push a 32-bit zero to the stack.

**0<** forth 0 ( n -- flg )
Leave a true flag if n is negative; otherwise leave a false flag.

**0=** forth 0 ( n -- flg )
Leave a true flag if n is equal to zero; otherwise leave a false flag.

**0branch** system 0 ( flg -- )
The run-time word compiled by if and other conditionals. When executed, if the flag is zero, execution branches to the address specified by the 16 bits which follow.

**0sector#** disking 49 ( -- adr )
Return the address of a parameter which tells whether the sectors on the disk in the current drive are numbered from 0 or 1.

**1+** forth 0 ( n -- n+1 )
Increment n by one.

copyright 1983 

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| **1-** | forth | 0 | ( n -- n-l ) |
| | Decrement n by one. | | |
| **1pass** | editor | 16 | ( from to cnt -- fr2 to2 ) |
| | Used by copies to copy as many blocks as available memory will hold. | | |
| **2!** | forth | 0 | ( d adr -- ) |
| | Store d at adr. | | |
| **2*** | forth | 0 | ( nl -- n2 ) |
| | Multiply nl by 2. Arithmetic shift left. | | |
| **2+** | forth | 0 | ( n -- n+2 ) |
| | Increment n by two. | | |
| **2-** | forth | 0 | ( n -- n-2 ) |
| | Decrement n by two. | | |
| **2/** | forth | 0 | ( nl -- n2 ) |
| | Divide nl by 2. Arithmetic shift right. | | |
| **2>r** | forth | 0 | ( nl n2 -- ) |
| | Transfer nl and n2 to the return stack. n2 is the most accessible after the transfer. Should be paired with 2r> in the same definition. | | |
| **2@** | forth | 0 | ( adr -- d ) |
| | Leave on the stack the 32-bit value at adr. | | |
| **2constant** | forth | 25 | ( d -- ) |
| | Define a 32-bit constant. When the defined constant is executed, d is pushed to the stack. | | |
| **2drop** | forth | 0 | ( nl n2 -- ) |
| | Drop the top two numbers from the parameter stack. | | |
| **2dup** | forth | 0 | ( d -- d d ) |
| | Copy the 32-bit number on top of the stack. | | |
| **2over** | forth | 11 | ( dl d2 -- dl d2 dl ) |
| | Leave a copy of the second double number on the stack. | | |
| **2r>** | forth | 0 | ( -- n2 nl ) |
| | Transfer nl and n2 from the return stack to the parameter stack. nl was the most accessible on the return stack prior to this operation. | | |

**WORD — VOCABULARY — BLOCK — STACK EFFECT**

**2rot** — forth — 11 — ( d1 d2 d3 -- d2 d3 d1 )
Rotate the third double number to the top of the stack.

**2swap** — forth — 11 — ( d1 d2 -- d2 d1 )
Exchange the top two double numbers on the stack.

**2variable** — forth — 25 — ( -- )
Define a 32-bit variable which is initialized to zero. When the defined variable executes, it pushes its address to the stack.

**:** — forth — 41 — ( -- )
Used in the form : xx ... ; to create a new word with the name xx . The words represented by ... determine the behavior of xx when it is subsequently executed.

**;** — forth — 41 — ( -- )
Terminate a colon definition and resume interpretation.

**;code** — forth — 9 — ( -- )
Used in the definition of a defining word to specify the run time behavior of the defined words as being the machine code compiled by the assembler words which follow.

**<** — assembler — 0 — ( -- )
Specify direct page addressing mode.

**<** — forth — 0 — ( n1 n2 -- flg )
Leave a true flag if n1 is less than n2; otherwise leave a false flag.

**<#** — forth — 0 — ( -- )
Initialize pictured numeric output conversion.

**<cmove** — forth — 0 — ( adr1 adr2 u -- )
Move u bytes from adr1 to adr2, the byte at adr1+u-1 is moved first.

**<lfa** — forth — 17 — ( cfa -- lfa )
Convert a word's execution address to the address of its link field.

**<nfa** — forth — 17 — ( cfa -- nfa )
Convert a word's execution address to the address of its count byte.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| <of | forth | 38 | ( -- adr ) |

Begins a phrase to be executed if the case select value is less than the number on the stack; otherwise execution branches to the words following the associated "else". See (<of).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| = | forth | 0 | ( nl n2 -- flg ) |

Leave a true flag if nl is equal to n2; otherwise leave a false flag.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| > | forth | 0 | ( nl n2 -- flg ) |

Leave a true flag if nl is greater than n2; otherwise leave a false flag.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >Drive | disking | 50 | ( dr# -- ) |

Set the current drive to be the specified drive.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >binary | forth | 0 | ( dl adrl -- d2 adr2 ) |

Convert the text at adrl+l to a binary value using the current base. The new value is added to dl and left as d2. adr2 is the address of the first non-convertible character. Set ctr equal to the number of converted digits.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >f | editor | 20 | ( -- ) |

Move the text which follows the editing command being executed to the find buffer. Do nothing if no text follows.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >i | editor | 20 | ( -- ) |

Move the text which follows the editing command being executed to the insert buffer. Do nothing if no text follows.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >in | forth | 0 | ( -- adr ) |

A user variable which holds the offset into the buffer (terminal or disk) from which the interpreter will fetch the next word.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >of | forth | 38 | ( -- adr ) |

Begins a phrase to be executed if the case select value is greater than the number on the stack; otherwise execution branches to the words following the next else . See (>of).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| >r | forth | 0 | ( n -- ) |

Transfer n to the return stack. Should be followed by r> in the same definition.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**?** forth 0 ( adr -- )
Print the 16-bit value at adr.

**?comp** forth 39 ( -- )
Aborts if FORTH is not in the compiling state.

**?cr** forth 15 ( cnt -- )
Issue a carriage return if there is not enough room on the current line of the current output device for the specified number of characters.

**?do** forth 40 ( -- adr1 adr2 )
Use only in a definition. Marks the beginning of a definite loop which must be terminated by loop or +loop. See (?do).

**?dup** forth 0 ( n -- n | n -- n n )
Duplicate the top of the stack if it is non-zero.

**?found** editor 21 ( flg -- )
If the flag is non-zero, print the text in the find buffer and an error message and execute quit.

**?leave** forth 40 ( -- )
An immediate word which compiles code to force immediate termination of a loop at run-time if the top of the stack is non-zero; otherwise execution continues. Must be used in a definition and within a loop. See (?leave).

**?loop** forth 40 ( -- )
Abort and issue an error message if a loop is not being compiled.

**?next** system 37 ( ??? flg -- ??? )
Used by run-time case words to control execution depending upon whether the case was matched. Stack effect varies depending on whether there was a match and whether the case select value is a number or a string.

**?pairs** forth 39 ( n1 n2 -- )
Used in "secure" versions of program structuring words to check syntax. Abort if n1 and n2 are not equal.

**?stack** forth 0 ( -- )
Abort if the parameter stack is in an underflow condition. Can only be used in a definition.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**?status** system 3 ( -- )
Aborts and issues an error message if the last disk access resulted in an error.

**@** forth 0 ( adr -- n )
Leave on the stack the 16-bit value at adr.

**BackUp** disking 52 ( FromDr# ToDr# -- )
Copy all blocks on the disk in the source drive to the disk in the destination drive.

**Bounds** disking 50 ( org limit -- )
Return the block number of the first block on the current drive and the number of the first block on the next drive.

**Claim** disking 53 ( cnt -- )
Claim the specified number of blocks on the disk in the current drive for use by eForth. The disk must be freshly formatted.

**ClearDisk** disking 52 ( -- )
Wipe all claimed blocks on the disk in the current drive.

**Configure** disking 50 ( -- )
Set the track and sector parameters for the current drive to those for which the current disk in the drive was formatted.

**Drive** disking 50 ( -- adr )
Return the address of a pointer to the parameters for the current drive.

**Drive0** disking 50 ( -- adr )
Return the address of the parameters for drive 0.

**DriveField** disking 49 ( b -- )
Create a name for a field in a drive's parameter field.

**Entries** disking 51 ( -- cnt )
Return the number of entries in the SectorCounts table.

**FormFeed** system 33 ( -- )
Emits an ascii form-feed. Installed in the 'page vector of the system printer.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| **I'm** | **forth** | **43** | **( -- )** |
| | Used in the form: I'm cee to place the user's initials into the variable me . | | |
| **Mark** | **editor** | **43** | **( -- )** |
| | Mark the current editing block with the time, the user's initials and the date. | | |
| **Mount** | **disking** | **53** | **( dr# -- )** |
| | Mount the disk in the specified drive. The disk must have been claimed previously. | | |
| **ReadSector** | **disking** | **52** | **( adr dadr -- )** |
| | Read the sector at dadr on the disk in the current drive storing it at adr. The high byte of dadr specifies the track; the low byte the sector. | | |
| **Release** | **disking** | **53** | **( cnt -- )** |
| | Claim all blocks on the disk in the current drive for use by eFORTH except for the specified number which are reserved for the system's operating system. The disk must be freshly formatted. | | |
| **Remove** | **disking** | **52** | **( dr# -- )** |
| | Remove the specified drive from the system. If drive 0 is specified, block 0 will access the first block on the disk in drive 1. | | |
| **Restore** | **disking** | **52** | **( -- )** |
| | Restore the head on the current drive | | |
| **SectorCounts** | **disking** | **51** | **( -- adr )** |
| | Return the address of a table which contains the sectors per side for each common count of sectors per track. | | |
| **SetDate** | **forth** | **4** | **( -- )** |
| | Set the current date to the string that follows. | | |
| **SetSides** | **disking** | **51** | **( sectors -- )** |
| | Set the s/s field for the current drive given the specified number of sectors per track. | | |
| **SetTime** | **forth** | **5** | **( -- )** |
| | Set the current time to the string that follows. | | |
| **Size** | **disking** | **50** | **( -- cnt )** |
| | Return the number of bytes required for the parameters for each drive. | | |

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**WriteSector** disking 52 ( adr dadr -- )
Write the data at adr onto the sector specified by dadr on the disk in the current drive. The high byte of dadr specifies the track; the low byte the sector.

**[** forth 0 ( -- )
Suspend compilation and begin interpretation.

**[']** forth 0 ( -- )
Compile the execution address of the next word in the definition as a literal. At run-time, that address is pushed to the stack. May only be used in a definition.

**[compile]** forth 0 ( -- )
Compile the execution address of the immediate word which follows instead of executing it. The immediate word will execute when the defined word executes.

**]** assembler 0 ( ??? -- ??? )
Specify the indirect addressing mode. Stack effect varies depending on the previously specified addressing mode, if any.

**]** forth 0 ( -- )
Enter the compiling mode.

**a** assembler 0 ( -- )
Specify the A accumulator as an operand of the subsequent psh, pul, tfr, or exg instruction.

**a** editor 23 ( -- )
Append the string which follows to the current line.

**a,s** assembler 0 ( -- post )
Specify the addressing mode of A accumulator offset from the S register. The appropriate post byte is left on the stack.

**a,u** assembler 0 ( -- post )
Specify the addressing mode of A accumulator offset from the U register. The appropriate post byte is left on the stack.

**a,x** assembler 0 ( -- post )
Specify the addressing mode of A accumulator offset from the X register. The appropriate post byte is left on the stack.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**a,y** assembler 0 ( -- post )

Specify the addressing mode of A accumulator offset from the Y register. The appropriate post byte is left on the stack.

**abort"** forth 0 ( -- )

An immediate word which compiles code so that at run-time, if the top of the stack is non-zero, the text which follows is printed and quit is executed. See (abort) .

**abs** forth 0 ( n -- u )

Return the absolute value of n.

**again** assembler 8 ( adr -- )

Compile an unconditional branch to the machine code at the specified address.

**again** forth 39 ( adr -- )

Use only in a definition. Compiles an unconditional branch back to the code marked with the previous "begin".

**allot** forth 0 ( n -- )

Reserve n bytes of space in the dictionary starting at the current address returned by here.

**and** forth 0 ( u1 u2 -- u3 )

Leave the bitwise logical and of u1 with u2.

**ascii** forth 13 ( -- c )

Return the ASCII code of the following character. If compiling, remove it from the stack and compile it as a literal.

**assembler** forth 0 ( -- )

Make the assembler vocabulary the context vocabulary.

**at** editor 18 ( -- adr rem )

Return the buffer address of the current cursor position in the current editing block and the number of characters remaining in the current line.

**at0** editor 18 ( -- adr c/l )

Set the cursor at the start of the current line and return its buffer address and the length of the line.

**b** assembler 0 ( -- )

Specify the B accumulator as an operand of the subsequent psh, pul, tfr, or exg instruction.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**b** editor 20 ( -- )
Make the previous block the current editing block.

**b,s** assembler 0 ( -- post )
Specify the addressing mode of B accumulator offset from the S register. The appropriate post byte is left on the stack.

**b,u** assembler 0 ( -- post )
Specify the addressing mode of B accumulator offset from the U register. The appropriate post byte is left on the stack.

**b,x** assembler 0 ( -- post )
Specify the addressing mode of B accumulator offset from the X register. The appropriate post byte is left on the stack.

**b,y** assembler 0 ( -- post )
Specify the addressing mode of B accumulator offset from the Y register. The appropriate post byte is left on the stack.

**b/blk** forth 7 ( -- u )
A system constant which returns the number of bytes in a block. This implementation returns a value of 1024.

**back** system 36 ( adr -- )
Compiles a branch vector back to the address on the stack.

**base** forth 0 ( -- adr )
Return the address of the user variable which holds the base which is being used for input and output conversion of numbers.

**begin** assembler 39 ( -- adr )
Push the current value returned by here to the stack. Used to mark the destination of a subsequent branch instruction.

**begin** forth 39 ( -- adr )
Use only in a definition. Used to mark the beginning of either a "begin..while..repeat" or "begin..again" or "begin..until" loop.

**bell** forth 0 ( -- )
Executes the word whose execution address is in the variable 'bell. Its initial value is (bell).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**bl** — forth — 0 — ( -- bl )
A constant which returns the code for an ASCII blank.

**blank** — forth — 0 — ( adr u -- )
Fill memory beginning at adr with a sequence of u blanks. If u is zero, no action is taken.

**blk** — forth — 0 — ( -- adr )
A user variable which holds the number of the block being interpreted. If this number is zero, input is being taken from the terminal input buffer.

**blk?** — system — 0 — ( u -- adr ff | u -- u u )
Search the buffers for block u. If found return its address and a zero; otherwise leave u and return a non-zero value.

**block** — forth — 0 — ( blk -- )
Leave the address of the first data byte in the disk buffer which contains block blk. The block is read from disk if necessary.

**blocks** — disking — 49 — ( -- adr )
Retur the address of the parameter which tells how many blocks are on the disk in the current drive.

**body** — forth — 17 — ( cfa -- pfa )
Convert a word's execution address to its parameter field address.

**bra** — assembler — 8 — ( adr -- )
Compile the machine code for a branch to the address on the stack. Compiles a long branch instruction if necessary.

**branch** — system — 0 — ( -- )
The run-time word compiled by repeat and other conditionals. When executed, causes execution to branch to the address specified by the 16 bits which follow.

**bs** — forth — 0 — ( -- )
Executes the word whose execution address is in the variable 'bs. Its initial value is (bs).

**bsr** — assembler — 8 — ( adr -- )
Compile the machine code for a branch to subroutine at the address on the stack. Compiles a long branch to subroutine if necessary.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**buf?** system 0 ( u -- adr flg )
Assign a buffer to block u. Return its address and a zero flag if the buffer is not marked as updated; otherwise the flag is the number of the updated buffer.

**buffer** forth 0 ( blk -- )
Obtain the next block buffer assigning it to block blk. The block is not read from disk.

**c!** forth 0 ( n adr -- )
Store the least significant 8-bits of n at adr.

**c#** forth 0 ( -- adr )
Return the address within the current output device record which contains the number of characters which have been printed on the current line.

**c,** forth 0 ( b -- )
Allot one byte of dictionary space and store the low byte of the number on the stack into it.

**c/l** forth 7 ( -- u )
A system constant which returns the number of characters on one line of an editing block. This implementation returns a value of 64.

**c@** forth 0 ( adr -- b )
Leave on the stack the 8-bit value at adr.

**case** forth 38 ( -- nl n2 )
Use only in a definition. Begins a keyed case structure.

**cc** assembler 9 ( -- cond )
Specify the "carry-clear" condition code.

**ccr** assembler 0 ( -- )
Specify the condition code register as an operand of the subsequent psh, pul, tfr, or exg instruction.

**center** forth 31 ( adr cnt -- )
Print the specified string at the center of the current print line.

**cfa>** forth 17 ( nfa -- cfa )
Convert the address of a word's count byte to its execution address.

**clear** editor 16 ( blk -- )
Fill the specified block with blanks.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**clears** editor 16 ( blk cnt -- )
Fill the specified range of blocks with blanks.

**cmove** forth 0 ( adrl adr2 u -- )
Move u bytes from adrl to adr2. The byte adr adrl is moved first.

**cnt** forth 0 ( -- adr )
A user variable used as a character limit for i/o operations.

**code** forth 9 ( -- )
Used to create a word whose behavior is specified with the machine code compiled by the assembler words which follow.

**compile** forth 0 ( -- )
Compile the execution address of the next word into the dictionary.

**constant** forth 0 ( n -- )
Used in the form n constant cc to create a named constant value. cc is added to the dictionary, and when it is executed n is pushed to the stack.

**context** forth 0 ( -- adr )
A user variable which specifies the context vocabulary.

**copies** editor 16 ( from to cnt -- )
Copy the specified number of blocks beginning at "from" moving them to "to".

**copy** editor 16 ( old new -- )
Copy the contents of the old block to the new block.

**count** forth 0 ( adr -- adr+l cnt )
Given the address of a string's character count, return the address of the first character and the length of the string.

**cr** forth 0 ( -- )
Executes the word whose execution address is in the current output device variable 'cr. See (cr).

**create** forth 0 ( -- )
Execute the word whose execution address is in the system variable 'create. Its initial value is (create).

copyright 1983 

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**cs** assembler 9 ( -- cond )
Specify the "carry-set" condition code.

**csp** forth 39 ( -- adr )
A user variable which holds the current stack position. Set by the colon and checked by the semicolon ("secure" versions only) to make sure that compiling did not change the stack.

**ctr** forth 0 ( -- adr )
A user variable used as a counter for i/o operations.

**current** forth 0 ( -- adr )
A user variable which specifies the current vocabulary.

**d** assembler 0 ( -- )
Specify the D register as an operand of the subsequert psh, pul, tfr, or exg instruction.

**d** editor 23 ( -- )
Delete the string which follows.

**d+** forth 25 ( d1 d2 -- d3 )
Return the 32-bit sum of d1 with d2.

**d,s** assembler 0 ( -- post )
Specify the addressing mode of D accumulator offset from the S register. The appropriate post byte is left on the stack.

**d,u** assembler 0 ( -- post )
Specify the addressing mode of D accumulator offset from the U register. The appropriate post byte is left on the stack.

**d,x** assembler 0 ( -- post )
Specify the addressing mode of D accumulator offset from the X register. The appropriate post byte is left on the stack.

**d,y** assembler 0 ( -- post )
Specify the addressing mode of D accumulator offset from the Y register. The appropriate post byte is left on the stack.

**d-** forth 26 ( d1 d2 -- d3 )
Leave the difference of two signed, 32-bit numbers.

**WORD** **VOCABULARY** **BLOCK** **STACK EFFECT**

**d.** **forth** **28** **( d -- )**
Print double number d followed by one space.

**d.r** **forth** **28** **( d u -- )**
Print double number d right-adjusted in a field which is u bytes wide.

**d0=** **forth** **26** **( d -- flg )**
Leave a true flag if d is equal to zero; otherwise return a false flag.

**d<** **forth** **26** **( d1 d2 -- flg )**
Leave a true flag if d1 is less than d2; otherwise leave a false flag.

**d=** **forth** **26** **( d1 d2 -- flg )**
Leave a true flag if two double numbers are equal; otherwise return a false flag.

**d>** **forth** **26** **( d1 d2 -- flg )**
Leave a true flag if d2 is greater than d1; otherwise leave a false flag.

**dabs** **forth** **26** **( d1 -- d2 )**
Leave the absolute value of a 32-bit number.

**date** **forth** **4** **( -- adr cnt )**
Convert the system date to a string.

**decimal** **forth** **0** **( -- )**
Set the input/output numeric conversion base to ten.

**definitions** **forth** **0** **( -- )**
Make the current vocabulary the same as the context vocabulary.

**delete** **editor** **21** **( -- )**
Delete the string which was just found with one of the searching commands.

**depth** **forth** **15** **( -- u )**
Return the number of lines per page on the current output device.

**disk** **system** **0** **( -- adr )**
Return the base address of the system disk parameters.

**disking** **system** **48** **( -- )**
Make the disking vocabulary the context vocabulary.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**dlv** system 0 ( -- adr )
A user variable used during the compiling of loops.

**dmax** forth 26 ( d1 d2 -- d3 )
Leave the highest of the two signed double numbers.

**dmin** forth 26 ( d1 d2 -- d3 )
Leave the lowest of the two signed double numbers.

**dnegate** forth 25 ( d1 -- -d1 )
Leave the two's complement of a 32-bit number.

**do** forth 40 ( -- adr1 adr2 )
Use only in a definition. Marks the beginning of a definite loop which must be terminated by loop or +loop. See (do).

**does>** forth 0 ( -- )
Used in the definition of a defining word. Terminates the words to be executed when the defining word executes and begins the words to be executed when the words defined with the new defining word are executed.

**dpl** forth 29 ( -- adr )
A user variable which gives the number of digits to the right of the last punctuation character in the last double number seen by the interpreter. A negative value indicates that the last number was not punctuated.

**dpr** assembler 0 ( -- )
Specify the direct page register as an operand of the subsequent psh, pul, tfr, or exg instruction.

**drcode** disking 49 ( -- adr )
Return the address of the system dependent drive code for the current drive.

**drop** forth 0 ( n -- )
Drop the top number from the stack.

**du<** forth 26 ( ud1 ud2 -- flg )
Leave a true flag if ud1 is less than ud2; otherwise leave a false flag. This is an unsigned comparison.

**dump** forth 10 ( adr cnt -- )
Print a memory dump of the specified number of bytes beginning at the specified address.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**dup** forth 0 ( n -- n n )
Leave a copy of the number on top of the stack.

**e** editor 23 ( -- )
Delete the string which was just found with one of the searching commands.

**editor** forth 6 ( -- )
Make the editor vocabulary the context vocabulary.

**else** assembler 8 ( adr1 -- adr2 )
Compile an unconditional branch leaving the address of the byte offset which later must be resolved, then resolve the branch at adr1 so that its target will be the code which follows.

**else** forth 39 ( adr1 -- adr2 )
Use only in a definition. Marks the end of the "if-true" phrase, and marks the beginning of the "if-false" phrase.

**emit** forth 0 ( c -- )
Executes the word whose execution address is in the variable 'emit. Its initial value is (emit).

**empty** forth 0 ( -- )
Removes all words from the user's dictionary space.

**empty-buffers** forth 0 ( -- )
Mark all block buffers as empty. Updated blocks are not written to disk and their modifications will be lost.

**end-code** forth 9 ( -- )
Used to terminate code and ;code definitions.

**endcase** forth 38 ( n1 n2 ??? -- )
Use only in a definition. Terminates a case structure. Stack effect varies according to the number of cases.

**eol** forth 15 ( -- )
Executes the word whose execution address is in the current output device variable 'eol. See (eol).

**eos** forth 15 ( -- )
Executes the word whose execution address is in the current output device variable 'eos. See (eos).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**eq** assembler 0 ( -- cond )
Specify the "z-bit-set" condition code.

**erase** forth 0 ( adr u -- )
Fill memory beginning at adr with a sequence of u nulls. If u is zero, no action is taken.

**execute** forth 0 ( cfa -- )
Execute the word whose execution address is on the stack.

**exit** forth 0 ( -- )
When used in a colon definition, execution of that definition will stop at that point and return to the calling word. When used on a load block, will terminate loading at that point and return to the calling word.

**expect** forth 0 ( adr cnt -- )
Executes the word whose execution address is in the system variable 'expect. Its initial value is (expect).

**f** editor 23 ( -- )
Starting at the editing cursor position, "find" the string which follows. Aborts if the string is not found.

**false** forth 11 ( -- ff )
Leave the constant which represents a boolean false.

**fill** forth 0 ( adrl u b -- )
Fill memory beginning at adr with a sequence of u copies of b. If u is zero, no action is taken.

**find** system 0 ( adr -- adr ff | cfa b )
Search the dictionary for the string at adr. Leave adr and return a zero if not found; otherwise leave the word's execution address under its count byte.

**first** system 0 ( -- adr )
A system constant which returns the beginning address of the system disk buffer area.

**flush** forth 0 ( -- )
Write all blocks to disk that have been flagged as updated.

**footer** forth 31 ( -- )
Move to the bottom line of the current page and print the system copyright message, then move to the top of the next page.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**forget** forth 0 ( -- )
Used in the form forget www to remove www ( and all words defined since www was defined) from the dictionary.

**forth** forth 0 ( -- )
Make the forth vocabulary the context vocabulary.

**forward** system 0 ( -- adr )
Mark the location of a forward branch which should subsequently be resolved with resolve .

**fxp** forth 28 ( -- adr )
A user variable which contains the maximum number of digits which may occur to the right of the last punctuation character in a double number. Numbers with fewer digits will be scaled. A negative value disables this feature.

**g** editor 22 ( blk line -- )
Get the specified line and insert it under the current line which then becomes the current line.

**ge** assembler 9 ( -- cond )
Specify the "greater-than-or-equal" condition code.

**gets** editor 22 ( blk line cnt -- )
Get the specified lines from the specified block and insert them under the current line. The current line becomes the last line inserted.

**golden** system 0 ( -- adr )
Return the address of the last saved dictionary state. The data at this address is used by empty .

**gt** assembler 0 ( -- cond )
Specify the "greater-than" condition code.

**h** forth 0 ( -- adr )
A user variable which contains the address of the next free byte in the dictionary.

**h0** forth 0 ( -- adr )
A user variable that contains the dictionary origin. Used by empty to re-origin the dictionary.

**header** forth 31 ( -- )
Print the system header on the next page.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**here** — **forth** — **0** — **( -- adr )**
Return the address of the next free byte in the dictionary.

**hex** — **forth** — **0** — **( -- )**
Set the input/output numeric conversion base to sixteen.

**hi** — **assembler** — **0** — **( -- cond )**
Specify the "hi" condition code.

**hold** — **forth** — **0** — **( c -- )**
Insert c into the pictured numeric output string.
Must be used between <# and #>.

**home** — **forth** — **15** — **( -- )**
Executes the word whose execution address is in the current output device variable 'home. See (home).

**hs** — **assembler** — **9** — **( -- cond )**
Specify the "higher-than-or-same" branch condition code.

**i** — **editor** — **23** — **( -- )**
Insert the string which follows at the cursor.

**i** — **forth** — **0** — **( -- index )**
Return the current loop index to the parameter stack.
Must only be used within a loop.

**id** — **forth** — **17** — **( nfa -- adr cnt )**
Convert a word's count byte address to a string which can be used by type. The string is placed at pad.

**id.** — **forth** — **17** — **( nfa -- )**
Print the name of the word whose count byte address is given on the stack. Issue a carriage return if it will not fit on the remainder of the current output line.

**if** — **assembler** — **8** — **( cond -- adr )**
Compile the machine code for a conditional forward branch (the condition is given on the stack). Leave the address of the relative offset which later must be resolved.

**if** — **forth** — **39** — **( -- adr )**
Use only in a definition. Marks the beginning of a phrase to be executed if the top of the stack is true; otherwise execution skips to the following else or then. At run-time, the top of the stack is removed.

**WORD** **VOCABULARY** **BLOCK** **STACK EFFECT**

**immediate** **forth** **0** **( -- )**
Mark the last defined word as an immediate word.

**in@** **forth** **0** **( -- adr )**
Return the address of the next character in the interpreter's input stream.

**index** **forth** **32** **( beg lim -- )**
Print the first lines of all blocks between beg and lim.

**input** **system** **0** **( adr -- )**
Make the specified device the current input device for the current user. Usage: term input

**insert** **editor** **21** **( -- )**
Insert the contents of the insert buffer at the position of the cursor.

**interpret** **forth** **0** **( -- )**
Interprets the input stream at the character indexed by >in until the input stream is exhausted.

**j** **forth** **0** **( -- index )**
Return the index of the next outer loop. May only be used within a loop within a loop in the same definition.

**k** **editor** **22** **( -- )**
Exchange the contents of the insert buffer with the contents of the find buffer. Allows the insertion of text which has been deleted.

**key** **forth** **0** **( -- c )**
Executes the word whose execution address is in the variable 'key. Its initial value is (key).

**key?** **forth** **0** **( -- flg )**
Execute the word whose execution address is in the system variable 'key?. Its initial value is (key?).

**l** **editor** **20** **( -- )**
List the current editing block

**l#** **forth** **0** **( -- adr )**
Return the address within the current output device record which contains the current line number on which the cursor or print head is positioned.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**last** forth 0 ( -- adr )
A user variable which contains the address of the count byte of the last word which was added to the dictionary.

**le** assembler 9 ( -- cond )
Specify the "less-than-or-equal" condition code.

**leave** forth 40 ( -- )
An immediate word which compiles code to force immediate termination of a loop at run-time. Must be used in a definition and must be used within a loop.

**limit** system 0 ( -- adr )
A system constant which returns the address of the end of the system disk buffer area.

**list** forth 10 ( blk -- )
List the specified block.

**listing** forth 32 ( blk -- )
Print the page on which the specified block falls.

**literal** forth 0 ( n -- )
An immediate word which compiles the number on the stack into the definition as a literal. At run-time, n is pushed to the stack.

**lo** assembler 0 ( -- cond )
Specify the "lo" condition code.

**load** forth 0 ( blk -- )
Begin interpretation of block blk. When finished, interpretation continues with the words following load.

**locate** forth 44 ( -- )
Used in the form: locate www to list the block from which www was loaded.

**loop** forth 40 ( adr1 adr2 -- )
Use only in a definition. Marks the end of a definite loop structure. See (loop).

**ls** assembler 9 ( -- cond )
Specify the "less-than-or-same" condition code.

**lt** assembler 0 ( -- cond )
Specify the "less-than" condition code.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**m*** forth 27 ( n1 n2 -- d )

Leave the signed 32-bit result of multiplying n1 by n2.

**m*/** forth 27 ( d1 n1 n2 -- d2 )

Multiply d1 by n1 leaving a 48-bit intermediate result which is then divided by n2 leaving a 32-bit result.

**m+** forth 27 ( d1 n -- d2 )

Leave the 32-bit result of adding n to d1. All values are signed.

**m/** forth 27 ( d n1 -- n2 )

Leave the signed 16-bit result of dividing d by n1.

**mark** system 36 ( -- adr )

Used by compiling words to mark the location of a backward reference.

**max** forth 0 ( n1 n2 -- n3 )

Leave the greater of the top two numbers on the stack.

**me** forth 43 ( -- adr cnt )

A string variable which contains the user's initials.

**mi** assembler 0 ( -- cond )

Specify the "n-bit-set" condition code.

**min** forth 0 ( n1 n2 -- n3 )

Leave the lesser of the top two numbers of the stack.

**mod** forth 0 ( u1 u2 -- u3 )

Unsigned divide of u1 by u2 leaving unsigned remainder u3.

**mon** system 0 ( -- )

Exit FORTH and return to the operating system.

**move** forth 0 ( adr1 adr2 u -- )

Move u bytes from adr1 to adr2. Unlike cmove and <cmove there is no danger of over-writing.

**n** editor 20 ( -- )

Make the next block the current editing block.

**ne** assembler 9 ( -- cond )

Specify the "not-equal" branch condition code.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|
| **negate** | **forth** | **0** | **( n -- -n )** |

Return the two's complement of n.

**next** **assembler** **0** **( -- )**

Compile the machine instructions which simulate the FORTH machine's "next" function. Must be used at each exit point in a code or ;code definition.

**noop** **system** **0** **( -- )**

This word performs no operation.

**not** **assembler** **0** **( n1 -- n2 )**

Negate the meaning of the preceeding condition code. For example, "eq not" is equivalent to "ne", and "cc not" is equivalent to "cs".

**not** **forth** **11** **( u1 -- u2 )**

Leave the one's complement of the number on the stack (each bit is inverted).

**number** **forth** **29** **( adr -- n or d )**

Convert the string whose count byte is at the specified address using the current base. Leaves a double number if the string is punctuated; otherwise leave a single number. The byte at adr is not used.

**of** **forth** **38** **( -- adr )**

Begins a phrase to be executed if the case select value equals the number on top of the stack; otherwise execution branches to the words following the next else . See (of) .

**ok** **forth** **33** **( -- )**

Make sure printer is positioned at the top of a page. If not, issue a formfeed and print the system footer.

**or** **forth** **0** **( u1 u2 -- u3 )**

Leave the bitwise logical or of u1 with u2.

**origin** **system** **0** **( -- adr )**

Return the base address of the system variable area.

**output** **system** **0** **( adr -- )**

Make the specified device the current output device for the current user. Usage: printer output

**over** **forth** **0** **( n1 n2 -- n1 n2 n1 )**

Leave a copy of the second number on the stack.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**p** **editor** **22** **( -- )**
Put the text which follows onto the current line.

**pad** **forth** **0** **( -- adr )**
Return the address of a scratch pad area which is 84 bytes above the address returned by here . Used to hold strings.

**page** **forth** **15** **( -- )**
Executes the word whose execution address is in the current output device variable 'page. See (page).

**pcr** **assembler** **0** **( -- )**
Specify the program counter as an operand of the subsequent psh, pul, tfr, or exg instruction.

**pick** **forth** **11** **( u -- n )**
Return the contents of the u-th stack value (not counting u itself). Undefined for u less than one. 2 pick is equivalent to over. 1 pick is equivalent to dup.

**pl** **assembler** **9** **( -- cond )**
Specify the "plus" branch condition code.

**prev** **system** **0** **( -- adr )**
A system variable which holds the address of the most recently accessed disk buffer.

**print** **forth** **33** **( -- )**
Redirect output to the system printer. All output of following words is sent to the printer.

**printer** **system** **0** **( -- adr )**
Device name for the system printer.

**protect** **system** **0** **( -- )**
Save the current state of the dictionary so that it can subsequently be restored by executing empty .

**ptr** **forth** **0** **( -- adr )**
A user variable used as a pointer for i/o operations.

**quit** **forth** **0** **( -- )**
Clear both stacks and return control to the terminal. No message is given to the user.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**r** — editor — 23 — ( -- )
Replace the string which was just found with the text which follows.

**r#** — forth — 0 — ( -- adr )
A user variable which contains the current character position (cursor) as an offset from the beginning of the current editing block.

**r/w** — forth — 0 — ( adr blk dir -- adr )
Executes the word whose execution address is in the variable 'r/w. Its initial value is (r/w).

**r0** — forth — 0 — ( -- adr )
A user variable that contains the address of the bottom of the return stack.

**r>** — forth — 0 — ( -- n )
Transfer n from the return stack to the parameter stack.

**r@** — forth — 0 — ( -- n )
Copy the top of the return stack onto the parameter stack.

**range** — forth — 38 — ( -- adr )
Begins a phrase to be executed if the case select value is "within" the numbers on the stack; otherwise execution branches to the associated "else". See (range).

**recurse** — forth — 36 — ( -- )
Compiles a recursive call to the word being defined.

**rel** — assembler — 0 — ( adr1 adr2 -- rel flg )
Return the relative offset between adr2 and adr1. Returns a true flag if it is greater than 8 bits wide.

**repeat** — assembler — 8 — ( adr1 adr2 -- )
Compile an unconditional branch back to adr1, and resolve the branch at adr2 to point to the code which follows.

**repeat** — forth — 39 — ( adr1 adr2 -- )
Use only in a definition. Marks the end of a "begin .. while .. repeat" structure.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**resolve** **system** **36** **( adr -- )**
Used by compiling words to resolve a forward reference located at the specified address.

**right** **forth** **31** **( adr cnt -- )**
Print the string at the specified address right adjusted on the current print line.

**roll** **forth** **11** **( u -- )**
Extract the u-th stack value to the top of the stack (not counting u itself) moving the remaining values into the vacated position. Undefined for u less than one. 3 roll is equivalent to rot. 1 roll is a null operation.

**rot** **forth** **0** **( n1 n2 n3 -- n2 n3 n1 )**
Rotate the top three values bringing the deepest to the top.

**s** **assembler** **0** **( -- )**
Specify the S register as an operand of the subsequent psh, pul, tfr, or exg instruction.

**s** **editor** **23** **( blk -- blk )**
Starting at the current editing block search for the string which follows through all blocks up to but not including the block specified on the stack. Aborts if the string is not found.

**s/b** **disking** **49** **( -- adr )**
Return the address of the parameter which tells how many sectors are required to hold one block on the disk in the current drive.

**s/s** **disking** **49** **( -- adr )**
Return the address of the parameter which tells how many sectors are on each side of the disk in the current drive.

**s0** **forth** **0** **( -- adr )**
A user variable that contains the address of the bottom of the stack and the start of the terminal input buffer.

**scan** **forth** **0** **( c adr1 -- adr2 cnt ) )**
Returns the starting address and count of the next word in the input stream delimited by the character "c".

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**scr** forth 0 ( -- adr )
Return the address of the user variable which holds the number of the current editing block.

**search** editor 21 ( -- )
Starting at the current cursor position, search for the string in the find buffer. Give an error message and abort if the string is not found.

**sectors** disking 49 ( -- adr )
Return the address of the parameter which tells how many sectors are on one trck of the disk in the current drive.

**show** forth 32 ( beg lim -- )
Print the documentation pages for all blocks between beg and lim.

**sign** forth 0 ( n d -- d )
Insert a minus sign into the pictured numeric output if n is negative. n is removed from the stack.

**space** forth 0 ( -- )
Transmit one ASCII blank to the current output device.

**spaces** forth 0 ( u -- )
Transmit u ASCII blanks to the current output device.

**speed** disking 49 ( -- adr )
Returns the address of the stepping speed for the current drive.

**state** forth 0 ( -- adr )
A user variable which if true means that a word is being compiled; otherwise the interpreter is executing each word in the input stream.

**string** forth 13 ( b -- )
Define a string variable which will hold strings up to a maximum of b bytes in length. When a word defined with string executes, it pushes the string's address to the stack and its maximum count.

**swap** forth 0 ( n1 n2 -- n2 n1 )
Exchange the top two stack values.

**sysI/O** system 0 ( -- adr )
Return the base address of the i/o vectors for the underlying system. This is the device "type" of term (the system terminal).

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**system** forth 0 ( -- )
Make the system vocabulary the context vocabulary.

**t** forth 18 ( line -- )
Make the specified line the current editing line.

**tab** forth 31 ( u -- )
Tab to column u. Backspace if column u is left of the current cursor position.

**term** system 0 ( -- adr )
Device name for the system terminal.

**text** forth 12 ( c -- )
Accept a string from the interpreter's input stream delimited by the character c and leave it at pad. pad is blank filled to 64 characters.

**then** assembler 8 ( adr -- )
Resolve the byte offset at the address on the stack so that the target of the branch will be the code which follows.

**then** forth 39 ( adr -- )
Use only in a definition. Marks the end of an "if-then" conditional structure.

**till** editor 23 ( -- )
Delete text from the cursor to (and including) the string which follows.

**time** forth 5 ( -- adr cnt )
Convert the system time, if any, to a string.

**tracks** disking 49 ( -- adr )
Return the address of the number of tracks on the current drive.

**true** forth 11 ( -- tf )
Leave the constant which represents a boolean true.

**type** forth 0 ( adr cnt -- )
Executes the word whose execution address is in the variable 'type. Its initial value is (type).

**u** assembler 0 ( -- )
Specify the U register as an operand of the subsequent psh, pul, tfr, or exg instruction.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**u** **editor** **22** **( -- )**
Move all following lines down (the last line is lost), then put the text which follows onto the line under the current line. Make the inserted line the current line.

**u*** **forth** **0** **( u1 u2 -- ud )**
Unsigned multiply of u1 by u2 leaving a 32-bit result.

**u.** **forth** **0** **( u -- )**
Print u followed by one space.

**u.r** **forth** **0** **( u1 u2 -- )**
Print u1 right adjusted in a field u2 characters wide.

**u/mod** **forth** **0** **( ud u1 -- u2 u3 )**
Divide double number ud by u1 leaving the remainder, u2, and the quotient, u3. All values are unsigned.

**u<** **forth** **0** **( u1 u2 -- flg )**
Leave a true flag if u1 is less than u2; otherwise leave a false flag. u1 and u2 are interpreted as unsigned 16-bit numbers.

**until** **assembler** **8** **( adr cond --**
Compile the machine code for a conditional branch (the condition is given on the stack) back to the address on the stack.

**until** **forth** **39** **( adr -- )**
Use only in a definition. Marks the end of a "begin..until" loop.

**update** **editor** **18** **( -- )**
Mark the current editing block as being updated. Optionally mark the block with the time, the user's initials, and the date.

**update** **forth** **0** **( -- )**
Mark the most recently referenced block buffer as modified. If the buffer is needed for another block, the modified block will be written to disk. Writing can be forced by executing flush.

**user** **forth** **0** **( u --)**
Create a name for a user variable which is offset u bytes above the base address of the user variable area. When the name is subsequently used, it returns the address of that user variable.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**v** **forth** **18** **( -- )**
Print the current line in the current editing block and show the cursor position.

**variable** **forth** **0** **( -- )**
Used in the form variable vvv to create a 16-bit variable. vvv is added to the dictionary and when executed, the address of the variable's 16-bit value is pushed to the stack.

**vc** **assembler** **0** **( -- cond )**
Specify the "v-bit-clear" condition code.

**version** **system** **7** **( )**
Returns the address of the system variable which holds a 32-bit base 36 number indicating the version.

**vocabulary** **forth** **0** **( -- )**
Used in the form vocabulary vvv to create a new vocabulary named vvv . When vvv is executed, it becomes the context vocabulary. When created, vvv is chained to the current vocabulary.

**vs** **assembler** **9** **( -- cond )**
Specify the "v-bit-set" branch condition code.

**while** **assembler** **8** **( cond -- adr )**
Compile the machine code for a conditional forward branch (the condition is given on the stack). Leave the address of the relative offset which later must be resolved.

**while** **forth** **39** **( -- adr )**
Use only in a definition. Marks the beginning of a phrase to be executed if, at run time, the top of the stack is non-zero. The phrase is terminated with repeat .

**width** **forth** **15** **( -- u )**
Return the number of characters per line on the current output device.

**wipe** **editor** **16** **( -- )**
Fill the current editing block with blanks.

**within** **forth** **11** **( n lo hi -- flg )**
Leave a true flag if n is less than hi and greater than or equal to lo; otherwise leave a false flag.

| WORD | VOCABULARY | BLOCK | STACK EFFECT |
|---|---|---|---|

**word** forth 0 ( c -- adr )

Reads the input stream until c is encountered. The text is placed at here with the character count in the first byte. Leading occurences of c are skipped.

**words** forth 17 ( -- )

List the words in the context vocabulary.

**x** assembler 0 ( -- )

Specify the **X** register as an operand of the subsequent psh, pul, tfr, or exg instruction.

**x** editor 22 ( -- )

Delete the current line moving all following lines up. The last line is filled with blanks.

**xor** forth 0 ( u1 u2 -- u3 )

Leave the bitwise logical exclusive or of u1 with u2.

**xy** forth 15 ( col row -- )

Executes the word whose execution address is in the current output device variable 'xy. See (xy).

**y** assembler 0 ( -- )

Specify the Y register as an operand of the subsequent psh, pul, tfr, or exg instruction.

**z** editor 22 ( -- )

Zip the cursor to the end of the text on the current line.

copyright 1983

# APPENDIX C

## eFORTH LISTINGS

This appendix contains listings of all eFORTH source blocks which are common to most eFORTH implementations. Listings for implementation specific source blocks are included with the documentation for the implementation.

**Block # 0**

```
 0 ( eFORTH SYSTEM DISK                         12:47pm cee 23jan84 )
 1
 2
 3                           eFORTH  Version 1.0
 4                           by Charles E. Eaker
 5
 6                 Distributed by Frank Hogg Laboratory, Inc.
 7                            The Regency Tower
 8                            770 James Street
 9                        Syracuse, New York 13203
10                             (315) 474-7856
11
12
13
14
15
```

**Block # 1**

```
 0 cr .(      eFORTH INITIAL PROGRAM LOAD     12:47pm cee 23jan84 )
 1 forth definitions decimal
 2      2 load | redefine (create) for locate utility
 3      3 load | install disk error trap
 4      6 load | eForth standard extensions
 5      4 load | system date
 6      5 load | system time
 7     18 load | eForth standard editor
 8     24 load | eForth double number electives
 9     30 load | eForth documentation electives
10     36 load | eForth compiler electives
11     42 load | eForth miscellaneous electives
12     48 load | eForth disking electives
13     60 load | hardware dependent electives
14     72 load | system dependent extensions
15 I'm cee          system protect  empty  decimal   exit
```

**Block # 2**

```
 0 ( create redefined for locate utility      12:47pm cee 23jan84 )
 1 (  This block redefines the behavior of the word executed
 2    by create.  It compiles the number of the block a word
 3    is loaded from as part of the word.  This number is used
 4    by locate to find and list the source block for the word.
 5    This means that each word requires two additional bytes
 6    of memory.  This feature can be disabled by simply not
 7    loading this block.  In that event, locate, on block 44,
 8    will not work properly. )
 9
10 system definitions
11 : (create)  ( -- )   blk @ , (create) ;
12
13     ' (create)  origin 20 + !    protect
14 forth definitions
15
```

Block # 3
```
 0 ( disk error trap                                12:47pm cee 23jan84 )
 1 system definitions hex
 2 : ?status ( -- )
 3     disk 2- @ ?dup if
 4     dup 80 and abort" Drives not ready."
 5     dup 40 and abort" Disk is write protected."
 6     dup 20 and abort" Write fault."
 7     dup 10 and abort" Sector not found on disk."
 8     dup 08 and abort" CRC error."
 9     dup 04 and abort" Lost data."
10     dup 02 and abort" Non-existent block."
11     then ;
12
13 : (r/w) ( adr blk dir -- adr )      (r/w)  ?status ;
14 decimal  ' (r/w) origin 14 + !   protect
15 forth definitions
```

Block # 4
```
 0 ( date SetDate                                   12:47pm cee 23jan84 )
 1
 2 8 string date ( -- adr cnt )
 3
 4 : SetDate ( -- ) bl word count drop date cmove ;
 5
 6    SetDate 23jan84 ( An example of how to set the date. )
 7
 8 exit
 9
10
11
12
13
14
15
```

Block # 5
```
 0 ( time SetTime                                   12:47pm cee 23jan84 )
 1
 2 8 string time ( -- adr cnt )
 3
 4 : SetTime ( -- ) bl word count drop time cmove ;
 5
 6    SetTime 12:47pm ( An example of how to set the time. )
 7
 8 exit
 9
10
11
12
13
14
15
```

**Block # 6**

```
 0 cr .(      eFORTH STANDARD EXTENSIONS     12:47pm cee 23jan84 )
 1
 2 vocabulary editor immediate    decimal
 3
 4    1 +load | system variables and constants
 5    2 +load | assembler conditionals
 6    3 +load | assembler extensions
 7    4 +load | ##  ####  dump  list  .s
 8    5 +load | stack and boolean extensions
 9    6 +load | string operations
10    8 +load | i/o extensions
11   10 +load | block editing operations
12   11 +load | header operations  words
13
14
15
```

**Block # 7**

```
 0 ( system constants and variables             12:47pm cee 23jan84 )
 1 system definitions
 2 origin 2+ dup constant version
 3      12 + dup constant 'r/w
 4        2+ dup constant 'start
 5        2+ dup constant 'number
 6        2+ dup constant 'create
 7        2+ dup constant 'key
 8        2+ dup constant 'key?
 9        2+ dup constant 'emit
10        2+ dup constant 'expect
11        2+ dup constant 'type
12        2+ dup constant 'bell
13        2+ dup constant 'bs    drop
14 forth definitions
15        64 constant c/l     1024 constant b/blk
```

**Block # 8**

```
 0 ( assembler conditionals                     12:47pm cee 23jan84 )
 1
 2 assembler definitions  hex
 3 : bsr ( adr -- )   here 1+ rel if 17 c, , else 8D c, c, then ;
 4 : bra ( adr -- )   here 1+ rel if 16 c, , else bl c, c, then ;
 5 : until  ( adr cond -- ) >r here 1+ rel
 6           if 1- 10 c, r> c, , else r> c, c, then ;
 7 : if   ( cond -- adr )    c, here 0 c, ;
 8 : then ( adr -- )   here over rel
 9          abort" branch too long." swap c! ;
10 : else   ( adr1 -- adr2 )  bl if swap then ;
11 : repeat  ( adr1 adr2 -- ) >r bra r> then ;
12 : again  ( adr -- )   bra ;
13 : while   ( cond -- adr ) if ;
14 forth definitions decimal
15
```

**Block # 9**

```
 0 ( assembler extensions                         12:47pm cee 23jan84 )
 1 ( branch conditions not defined in the pre-compiled portion. )
 2 assembler definitions
 3 : ne ( -- cond )   eq not ;    : pl ( -- cond )   mi not ;
 4 : ls ( -- cond )   hi not ;    : hs ( -- cond )   lo not ;
 5 : vs ( -- cond )   vc not ;
 6 : ge ( -- cond )   lt not ;    : le ( -- cond )   gt not ;
 7 : cc ( -- cond )   lo not ;    : cs ( -- cond )   cc not ;
 8
 9 forth definitions
10 : ;code   system compile (;code)
11           [compile] assembler r> drop ; immediate
12 : code    create here dup 2- ! [compile] assembler ;
13 : end-code current @ context ! ;
14
15
```

**Block # 10**

```
 0 ( ##  ####  dump  list  .s                     12:47pm     23jan84 )
 1
 2 : ## ( b -- ) base @ >r hex 0 <# # # #> type space r> base ! ;
 3 : #### ( u -- ) base @ >r
 4    hex 0 <#  # # # #  #> type space r> base ! ;
 5 : dump ( adr cnt --- )   base @ >r hex  cr 5 spaces
 6    over 16 0 do dup 15 and 45 emit . 1+ loop drop space
 7    over 16 0 do dup 15 and 1 .r 1+ loop drop r> base !
 8    over + swap do cr i ####  16 0 do i j + c@ ## loop space
 9      16 0 do i j + c@ 127 and dup bl < if drop 95 then emit loop
10    16 +loop ;
11 : list ( scr --- ) dup scr ! cr ." Block # " . b/blk c/l / 0
12    do cr i 2 .r space scr @ 0= i b/blk c/l / 1- = and ?leave
13     scr @ block i c/l * + c/l -trailing type loop cr ;
14 : .s ( print stack ) cr 's s0 @ 2- do i @ . -2 +loop ;
15
```

**Block # 11**

```
 0 ( stack and boolean extensions                 12:47pm cee 23jan84 )
 1 code roll  ( u -- )
 2   0 ,u ldd   0 ,u addd   d,u leax   0 ,x ldd   0 ,u std
 3   u pshs   begin   ,--x ldd   2 ,x std   0 ,s cmpx   eq  until
 4   2 ,u leau   2 ,s leas   next   end-code
 5 code pick  ( u -- n )
 6     0 ,u ldd   0 ,u addd   d,u ldd   0 ,u std   next  end-code
 7 code 2over   4 ,u ldd    6 ,u ldx    d x pshu   next end-code
 8 code 2swap   0 ,u ldd   4 ,u ldx   0 ,u stx   4 ,u std
 9       2 ,u ldd   6 ,u ldx   2 ,u stx   6 ,u std
10       next   end-code
11 : 2rot      >r >r 2swap r> r> 2swap ;
12 code not ( bool -- bool ) 0 ,u com   1 ,u com   next end-code
13 -1 constant true
14  0 constant false
15 : within ( n lo hi -- flg )    >r 1- over < swap r> < and ;
```

**Block # 12**

```
 0 ( string extensions                              12:47pm cee 23jan84 )
 1 : text ( c -- )
 2    pad c/l 2+ blank word pad over c@ 2+ cmove ;
 3
 4 code -text ( adr1 cnt adr2 -- flg )
 5     y pshs   0 ,u ldx
 6     2 ,u ldd   d,x leay    0 ,u sty   4 ,u ldy
 7     1 # bitb   eq not
 8     if   ,y+ lda   ,x+ suba   eq
 9     if   swap then
10     begin   0 ,u cmpx   eq not
11      if swap ,y++ ldd   ,x++ subd   eq not until
12      then then
13     4 ,u std   4 ,u leau   y puls
14     next   end-code
15 -->
```

**Block # 13**

```
 0 ( string extensions                              12:47pm cee 23jan84 )
 1 system definitions
 2 code (")     ( -- adr cnt )    | run-time word compiled by "
 3      ,y+ ldb   clra   d y pshu   d,y leay   next   end-code
 4
 5 forth definitions
 6 : ascii    ( -- )  | compile or interpret an ascii character
 7      bl word 1+ c@ state @  ( a "smart" word )
 8      if [compile] literal then ; immediate
 9 : " ( -- )  | compile or interpret a string literal
10      state @   ( a "smart" word )
11      if compile system (") ascii " word  c@ 1+ allot
12      else ascii " text pad count then ; immediate
13 : string ( b -- )  | create string variable of length b
14      create dup c, 0 do bl c, loop does> count ;
15
```

**Block # 14**

```
 0 ( i/o extensions                                 12:47pm cee 23jan84 )
 1 system definitions
 2  20 user 'put      | holds address of current output device
 3  22 user 'get      | holds address of current input device
 4
 5 : 'device  ( b -- )  create c, does> c@ 'put @ + ;
 6    12 dup 'device 'depth | address of device depth
 7    2+ dup 'device 'width | address of device width
 8    2+ dup 'device 'cr    | cr execution vector for this device
 9    2+ dup 'device 'page  | page "          "     "     "    "
10    2+ dup 'device 'home  | home cursor     "     "     "    "
11    2+ dup 'device 'xy    | position cursor       "     "    "
12    2+ dup 'device 'eol   | erase to end of line "     "    "
13    2+     'device 'eos   | erase to end of screen     "    "
14 forth definitions  -->
15
```

**Block # 15**

```
 0 ( i/o extensions                              12:47pm cee 23jan84 )
 1
 2 : width (    -- u )   system 'width @ ;
 3 : depth (    -- u )   system 'depth @ ;
 4 : xy    ( x y -- )    system 'xy @ execute ;
 5 : page  ( -- )        system 'page @ execute  0 l# !  0 c# ! ;
 6 : home  ( -- )        system 'home @ execute ;
 7 : eol   ( -- )        system 'eol @ execute ;
 8 : eos   ( -- )        system 'eos @ execute ;
 9
10 : ?cr   ( cnt -- )    width c# @ - > if cr then ;
11
12
13
14
15
```

**Block # 16**

```
 0 ( block editing operations                    12:47pm cee 23jan84 )
 1 editor definitions
 2 : copy   ( old new -- )  flush swap block 2- ! update ;
 3 : clear  ( blk --- )  block b/blk blank update ;
 4 : clears ( blk cnt -- )   0 ?do dup i + clear loop drop ;
 5 : wipe   ( -- ) scr @ clear ;
 6 : lpass ( from to cnt -- nextfrom nextto )
 7     here 4 pick 3 pick over + swap
 8     ?do i true r/w b/blk + loop drop
 9     here 3 pick 3 pick over + swap
10     ?do i false r/w b/blk + loop drop
11     rot over + rot rot + ;
12 : copies ( from to cnt -- )
13     's 256 - here - b/blk / dup >r /mod swap >r
14     0 ?do 'r 6 + @ lpass loop r> lpass 2drop r> drop ;
15 forth definitions
```

**Block # 17**

```
 0 ( header operations   words                   12:47pm cee 23jan84 )
 1 code <nfa  ( cfa -- nfa )
 2    x pulu   -1 ,x leax   begin   ,-x tst   mi   until
 3    x pshu   next   end-code
 4 code cfa>  ( nfa -- cfa )
 5    x pulu   ,x+ ldb   31 # andb   b,x leax   x pshu
 6    next   end-code
 7 : <lfa  ( cfa -- lfa )    <nfa 2- ;
 8 : body ( cfa -- pfa ) 2+ ;
 9 : id    ( nfa -- adr cnt )  count 31 and pad c! pad count cmove
10      pad dup c@ + dup c@ 127 and swap c! pad count ;
11 : id.   ( nfa -- )
12      id dup 1+ width c# @ - > if cr then type space ;
13 : words ( -- )
14      cr context @ @ begin ?dup while dup id. 2- @ repeat cr ;
15
```

**Block # 18**

```
 0 cr .(      eFORTH STANDARD EDITOR          12:47pm cee 23jan84 )
 1 editor definitions
 2 variable 'update          ' update 'update !
 3 : update 'update @ execute ;
 4
 5 : at ( -- adr rem ) r# @ dup b/blk 1- over u<
 6      abort" off of current editing screen."
 7      scr @ block + c/l rot c/l 1- and - ;
 8 : at0 ( -- adr c/l ) at c/l - r# +! drop at ;
 9
10 forth definitions
11 : v ( -- )  editor cr space
12     at 2dup c/l swap - dup >r - r> type 94 emit type
13     space r# @ c/l / . [compile] editor ;
14 : t ( n -- )  c/l * r# ! v ;
15 editor definitions   1 +load   forth definitions
```

**Block # 19**

```
 0 ( -match                                   12:47pm cee 23jan84 )
 1
 2 code -match ( adr1 cnt1 adr2 cnt2 -- adr3 flg )
 3     0 ,u ldd   d y pshs   6 ,u ldx   4 ,u ldd   bl if
 4     begin 6 ,u ldx   1 ,x leax   6 ,u stx   4 ,u ldd
 5     1 # subd   4 ,u std   swap then
 6     0 ,u cmpd   lo not if 0 ,u ldd   0 ,s std   2 ,u ldy
 7     begin ,y+ lda   ,x+ cmpa   rot eq until
 8     d puls   1 # subd   d pshs   eq until  clrb
 9     begin clra   4 ,u leau   0 ,u std   2 ,u stx   d y puls next
10     swap then 4 ,u ldd   6 ,u ldx   d,x leax   1 # ldb   bra
11     end-code
12
13 -->
14
15
```

**Block # 20**

```
 0 ( editor primitives                         12:47pm cee 23jan84 )
 1
 2 : l ( -- ) scr @ list ;
 3 : b ( -- ) -1 scr +! 0 r# ! ;
 4 : n ( -- )  1 scr +! 0 r# ! ;
 5 : #i ( -- adr ) pad c/l 2+ + ;
 6 : #f ( -- adr ) pad c/l 2+ 2* + ;
 7 : >i ( -- ) 94 text pad c@ if pad #i c/l 2+ cmove then ;
 8 : >f ( -- ) 94 text pad c@ if pad #f c/l 2+ cmove then ;
 9
10 -->
11
12
13
14
15
```

**copyright 1983** 

**Block # 21**

```
 0 ( insert delete and search primitives      12:47pm cee 23jan84 )
 1
 2 : insert ( -- )
 3     at dup #i c@ min dup >r - 0 max over dup r@ + rot <cmove
 4     #i 1+ swap r@ cmove r> r# +! update ;
 5 : delete ( -- )
 6     #f c@ >r r@ negate r# +!  at drop r@ + at r@ - 2dup + >r
 7     cmove  r> r> blank update ;
 8 : -search ( -- flg )
 9     at drop dup >r b/blk r# @ - 0 max #f count -match
10     swap r> - over if drop else r# +! then ;
11 : ?found ( flg -- )
12     if #f count type ."  ?" quit then ;
13 : search ( -- )
14     >f -search ?found ;
15 -->
```

**Block # 22**

```
 0 ( line editing commands                      12:47pm cee 23jan84 )
 1
 2 : x ( -- )  at0 -trailing #i c! #i count cmove
 3     at over + swap dup >r b/blk r# @ - c/l - dup >r cmove
 4     r> r> swap + c/l blank update ;
 5 : p ( -- ) at0 blank >i insert ;
 6 : u ( -- ) c/l r# +! at0 over + b/blk r# @ - c/l - <cmove p ;
 7 : g ( scr line -- ) c/l * swap block + c/l -trailing
 8     #i c! #i count cmove u ;
 9 : gets ( scr line cnt -- ) over + swap ?do dup i g loop drop ;
10 : z ( -- ) at0 -trailing r# +! drop ;
11 : k ( -- ) #i pad 132 cmove pad #f 66 cmove ;
12 -->
13
14
15
```

**Block # 23**

```
 0 ( string editing commands                    12:47pm cee 23jan84 )
 1
 2 : till ( -- ) >f at over >r #f count -match ?found r> -
 3     dup #f c! at drop #f count cmove r# +! delete v ;
 4
 5 : s ( scr -- scr )   >f 0 over scr @
 6     ?do drop -search dup 0= if v forth i . leave then n loop
 7     ?found ;
 8
 9 : f ( -- ) search v ;
10 : e ( -- ) delete v ;
11 : i ( -- ) >i insert v ;
12 : a ( -- ) z i ;
13 : r ( -- ) delete i ;
14 : d ( -- ) search e ;
15
```

copyright 1983 

Block # 24

```
 0 cr .(      eFORTH DOUBLE NUMBERS              12:47pm cee 23jan84 )
 1
 2 forth definitions
 3    1 +load | 2constant 2variable d+ dnegate
 4    2 +load | double number operations
 5    3 +load | mixed precision operations
 6    4 +load | double number output
 7    5 +load | double number input - interpretation only
 8
 9 exit
10
11
12
13
14
15
```

Block # 25

```
 0 ( 2variable  2constant  d+  dnegate          12:47pm cee 23jan84 )
 1
 2 : 2constant ( d -- )   create , , ;code  2 ,x ldd   4 ,x ldx
 3                                d x pshu   next   end-code
 4  0 0 2constant 0.
 5 : 2variable ( -- )    variable 0 , ;
 6 code d+  ( d1 d2 -- d3 )
 7               2 ,u ldd   6 ,u addd   6 ,u std
 8               0 ,u ldd   5 ,u adcb   4 ,u adca
 9               4 ,u std   4 ,u leau   next   end-code
10 code dnegate ( d1 -- -d1 )
11               clra   clrb   2 ,u subd   2 ,u std   0 # ldd
12               1 ,u sbcb   0 ,u sbca   0 ,u std
13               next end-code
14
15
```

Block # 26

```
 0 ( double number operations                   12:47pm cee 23jan84 )
 1
 2 : dabs ( d1 -- d2 )      dup 0< if dnegate then ;
 3 : d-   ( d1 d2 -- d3 )   dnegate d+ ;
 4 : d0=  ( d -- flg )      or 0= ;
 5 : d=   ( d1 d2 -- flg )  d- d0= ;
 6 : d<   ( d1 d2 -- flg )  d- swap drop 0< ;
 7 : d>   ( d1 d2 -- flg )  2swap d< ;
 8 : dmin ( d1 d2 -- d3 )   2over 2over d> if 2swap then 2drop ;
 9 : dmax ( d1 d2 -- d3 )   2over 2over d< if 2swap then 2drop ;
10
11 code du< ( ud1 ud2 -- flg )
12     4 ,u ldd   0 ,u cmpd
13     lo not  if   6 ,u ldd   2 ,u cmpd   then
14     0 # ldd   lo if   coma   comb   then
15     6 ,u leau   0 ,u std   next   end-code
```

**Block # 27**

```
 0 ( mixed precision arithmetic                  12:47pm cee 23jan84 )
 1
 2 : m+  ( d1 n -- d2 )    dup 0< d+ ;
 3 : m/ ( d n1 -- n2 )
 4     2dup xor >r abs >r dabs r> u/mod
 5     r> 0< if negate then swap drop ;
 6 : m* ( n1 n2 -- d )
 7     2dup xor >r abs swap abs u*
 8     r> 0< if dnegate then ;
 9 : */ ( n1 n2 n3 -- n4 )   >r m* r> m/ ;
10 : m*/  ( d1 n1 n2 -- d2 )    2 pick 4 pick xor >r
11     >r abs >r dabs r> 2>r r@ u* 0 2r> u* d+
12     r@ abs u/mod r@ abs swap >r u/mod r> rot drop
13     2r> xor 0< if dnegate then ;
14
15
```

**Block # 28**

```
 0 ( double number output                        12:47pm cee 23jan84 )
 1
 2 52 user fxp            -1 fxp !
 3
 4 : d.r  ( d u -- )
 5           >r swap over dabs <# fxp @ 0< 0=
 6           if fxp @ ?dup if 0 do # loop then ascii . hold
 7            begin 3 0 do 2dup or if # else leave then loop
 8             2dup or dup if ascii , hold then 0= until
 9           else #s then sign #> r> over - spaces type ;
10 : d.  ( d -- )
11           0 d.r space ;
12
13 exit
14
15
```

**Block # 29**

```
 0 ( double number input                         12:47pm cee 23jan84 )
 1
 2 54 user dpl
 3
 4 : number ( adr -- n or d )
 5     0 dpl !   dup 1+ c@ ascii - = dup >r - 0 0 rot
 6     begin >binary dup c@ bl -
 7      while dup c@ dup ascii : =
 8        swap ascii , ascii 0 within or dup 0= abort" ?" dpl !
 9      repeat  drop r> if dnegate then
10     dpl @ if cnt @ else drop -1 then dpl ! ;
11
12 system  ' number 'number !    protect
13
14
15
```

Block # 30

```
 0 cr .(      eFORTH DOCUMENTATION ELECTIVES  12:47pm cee 23jan84 )
 1
 2 forth definitions
 3    1 +load | tab  right  center  footer  header
 4    2 +load | index  listing  show
 5    3 +load | printer control words
 6 exit
 7
 8
 9
10
11
12
13
14
15
```

Block # 31

```
 0 ( tab   center   right  footer  header     12:47pm cee 23jan84 )
 1 : tab ( n -- )   c# @ - dup 0<
 2      if abs 0 do bs loop else spaces then ;
 3 : center ( adr cnt -- )  width 2/ over 2/ - tab type ;
 4 : right ( adr cnt -- ) width over - tab type ;
 5
 6 : footer ( -- )
 7     l# @ depth mod  depth 2- swap ?do cr loop
 8     cr ." copyright 1983"
 9     " Charles E. Eaker" right page ;
10
11 : header ( -- )   l# @ if footer then
12      cr cr time type
13      " eFORTH DOCUMENTATION" center
14      date right cr cr ;
15
```

Block # 32

```
 0 ( index  listing  show   3/page           12:47pm cee 23jan84 )
 1
 2 : index ( n1 n2 --- )
 3    swap  dup 60 mod if header then
 4    do i 60 mod if cr else header then
 5       i block i 5 .r space c/l -trailing  type
 6    loop cr ;
 7
 8 : listing ( blk -- )  header
 9     3 / 3 * dup 3 + swap do cr i list loop ;
10
11 : show ( beg end -- )  swap do i listing 3 +loop ;
12
13
14
15
```

**Block # 33**

```
 0 ( printer control words                    12:47pm cee 23jan84 )
 1 ( define and install printer form-feed and fancy cr )
 2 system definitions
 3 : FormFeed ( -- )   12 emit ;    ( define it )
 4 : (cr) ( -- ) key?
 5     if key 27 =
 6      if begin key? until key 13 = abort" aborted." then
 7     then (cr) 0 c# ! 1 l# +! ;
 8
 9 printer output ' FormFeed 'page !  ' (cr) 'cr !
10 term output ' (cr) 'cr !
11
12 forth definitions
13 : print ( -- ) system printer output ;
14 : ok ( -- ) footer ;
15
```

**Block # 34**

```
 0 ( print vocabularies                       12:47pm cee 23jan84 )
 1
 2 header
 3 .( FORTH VOCABULARY) forth words cr cr
 4 .( SYSTEM VOCABULARY) system words cr cr
 5 .( EDITOR VOCABULARY) editor words cr cr
 6 .( ASSEMBLER VOCABULARY) assembler words cr cr
 7
 8 exit
 9
10 To get a listing of words in the vocabularies, just load this
11 block.  To send it to the printer, just enter
12          print 40 load ok
13
14
15
```

**Block # 35**

```
 0 ( reserved                                 12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

copyright 1983 Frank Hogg Laboratory

Block # 36

```
 0 cr .(      eFORTH COMPILER ELECTIVES       12:47pm cee 23jan84 )
 1
 2 system definitions
 3 : resolve  ( adr -- )   here swap ! ;
 4 : mark     ( -- adr )   here ;
 5 : back     ( adr -- )   , ;
 6
 7 forth definitions
 8 : recurse ( -- )    | compile a recursive call
 9               last @ cfa> , ;   immediate
10
11     1 +load | positional case structure
12     3 +load | compiler security
13
14 exit
15
```

Block # 37

```
 0 ( keyed case run-time words                 12:47pm cee 23jan84 )
 1
 2 system definitions
 3 : ?next  ( used by case run-time words )   r> drop ?dup
 4     if 0< if drop else 2drop then r> 2+ else r> @ then >r ;
 5 : (of)     over = ?next ;
 6 : (<of)    over swap < ?next ;
 7 : (>of)    over swap > ?next ;
 8 : (range)  3 pick >r within r> swap ?next ;
 9 : ("of)    2over drop -text 0= negate ?next ;
10 forth definitions
11
12   -->
13
14
15
```

Block # 38

```
 0 ( keyed case compiling words                12:47pm cee 23jan84 )
 1
 2 : of        system compile (of) forward ;  immediate
 3 : <of       system compile (<of) forward ;  immediate
 4 : >of       system compile (>of) forward ;  immediate
 5 : range     system compile (range) forward ;  immediate
 6 : "of       system compile ("of) forward 1 ptr @ ! ; immediate
 7 : case      0 's ptr ! 0 ; immediate
 8 : endcase   compile drop ptr @ @ if compile drop then
 9             begin ?dup while system resolve repeat drop ;
10             immediate
11
12
13
14
15
```

**Block # 39**

```
 0 ( standard conditionals redefined          12:47pm cee 23jan84 )
 1 56 user csp
 2 : ?comp ( -- )  state @ 0= abort" Compilation only." ;
 3 : ?pairs ( n n -- )   ?comp  - abort" syntax error." ;
 4 : begin    ?comp [compile] begin 1 ; immediate
 5 : until    1 ?pairs [compile] until ; immediate
 6 : else     6 over = if drop [compile] else 5
 7            else 2 ?pairs [compile] else 2 then ; immediate
 8 : if       ?comp [compile] if 2 ; immediate
 9 : then     2 ?pairs [compile] then ; immediate
10 : while    ?comp [compile] while 4 ; immediate
11 : repeat   4 ?pairs >r 1 ?pairs r> [compile] repeat ; immediate
12 : again    1 ?pairs [compile] again ; immediate
13 assembler definitions
14 : begin    here ;  | The one above won't work in the assembler.
15 forth definitions    -->
```

**Block # 40**

```
 0 ( case and loop words redefined            12:47pm cee 23jan84 )
 1 : case     ?comp [compile] case 5 ; immediate
 2 : of       5 ?pairs [compile] of 6 ; immediate
 3 : <of      5 ?pairs [compile] <of 6 ; immediate
 4 : >of      5 ?pairs [compile] >of 6 ; immediate
 5 : range    5 ?pairs [compile] range 6 ; immediate
 6 : "of      5 ?pairs [compile] "of 6 ; immediate
 7 : endcase  6 ?pairs [compile] endcase ; immediate
 8 : do       ?comp [compile] do 3 ; immediate
 9 : ?do      ?comp [compile] ?do 3 ; immediate
10 : loop     3 ?pairs [compile] loop ; immediate
11 : +loop    3 ?pairs [compile] +loop ; immediate
12 : ?loop    system dlv @ 0= abort" must be used in a loop." ;
13 : leave    ?loop [compile] leave ; immediate
14 : ?leave   ?loop [compile] ?leave ; immediate
15 -->
```

**Block # 41**

```
 0 ( colon and semicolon redefined            12:47pm cee 23jan84 )
 1
 2 : :  state @ abort" execution only." 's csp ! : ; immediate
 3   ( The old version of the colon is not immediate. )
 4
 5 : ; ( -- ) ?comp 's csp @ - abort" incomplete definition."
 6         compile exit r> drop ; immediate
 7
 8 ( Redefine word executed by create to warn when a word is being
 9   redefined. )
10 system definitions
11 : (create)    >in @  bl word system find forth
12     if cr here count type ."  isn't unique." then drop >in !
13     (create) ;
14 ' (create) 'create !
15 system protect forth definitions
```

**Block # 42**
```
 0 cr .(      eFORTH MISCELLANEOUS ELECTIVES  12:47pm cee 23jan84 )
 1
 2      1 +load | block marking utility
 3      2 +load | locate utility
 4
 5 exit
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 43**
```
 0 ( block marking facility                   12:47pm cee 23jan84 )
 1 forth definitions
 2 4 string me  ( -- adr cnt )
 3 : I'm  ( -- )  bl text pad 1+ me cmove ;
 4 editor definitions
 5 : Mark ( -- ) | Mark block with id string
 6    scr @ block >r
 7    time    r@ c/l 21 - + swap cmove
 8    bl      r@ c/l 14 - + c!
 9    me      r@ c/l 13 - + swap cmove
10    date    r@ c/l 9 - + swap cmove
11    bl      r@ c/l 2-  + c!
12    ascii ) r> c/l 1- + c!
13    forth update ;
14 ' Mark   'update !     system protect
15 forth definitions
```

**Block # 44**
```
 0 ( locate utility                           12:47pm cee 23jan84 )
 1
 2 ( This word assumes that block 2 has been loaded. )
 3
 4 : locate ( -- )
 5    ' dup system ['] ?status < swap <lfa 2- @ dup 0= rot or
 6    abort" wasn't loaded."   list ;
 7
 8
 9
10
11
12
13
14
15
```

**Block # 45**

```
 0 ( reserved                                12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 46**

```
 0 ( reserved                                12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 47**

```
 0 ( reserved                                12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Block # 48

```
 0 cr .(      eFORTH DISKING ELECTIVES        12:47pm cee 23jan84 )
 1
 2 system definitions
 3 vocabulary disking immediate
 4 disking definitions
 5    1 +load | Drive table field names
 6    2 +load | disking primitives
 7    3 +load | Sectorcounts  SetSides
 8    4 +load | ClearDisk Remove Backup Restore ReadSector WriteSec
 9    5 +load | Claim  Release  Mount
10
11 forth definitions  exit
12
13
14
15
```

Block # 49

```
 0 ( drive parameter record fields             12:47pm cee 23jan84 )
 1
 2 : DriveField ( offset bytes -- offset ) | create field name
 3     create over c, + does> c@ disk 2- 2- @ + ;
 4
 5  0 2 DriveField blocks      | number of blocks.
 6    1 DriveField sectors     | number of sectors/track
 7    1 DriveField s/s         | sectors/side
 8    2 DriveField b/s         | bytes/sector
 9    1 DriveField s/b         | sectors/block
10    1 DriveField 0sector#    | first phys. sector # on track
11    1 DriveField tracks      | number of tracks
12    1 DriveField drcode      | physical drive code
13    1 DriveField speed       | stepping speed
14 drop
15
```

Block # 50

```
 0 ( drive drive0 >drive bounds                12:47pm cee 23jan84 )
 1
 2 variable 'claim ( -- adr )   ' 2drop  'claim !
 3 variable 'config ( -- adr )  ' noop  'config !
 4
 5 : Configure ( -- ) 'config @ execute ;
 6 : Size ( -- cnt )  disk 2+ c@ ;
 7 : Drive ( -- adr ) disk 2- 2- ;
 8 : Drive0 ( -- adr )  disk 3 + ;
 9 : >Drive ( dr# -- )    | Set current Drive.
10     dup 3 > abort" Non-existent drive."
11     disk 2+ count rot * + Drive ! ;
12 : Bounds ( -- org cnt )
13     0 Drive @ Drive0 ?do i @ + 16 +loop Drive @ @ ;
14
15
```

copyright 1983 

**Block # 51**

```
 0 ( SectorCounts  SetSides                    12:47pm cee 23jan84 )
 1
 2 create SectorCounts ( -- adr )
 3 ( 1 side        2 sides                               )
 4  10 c, 10 c,  20 c, 10 c, ( 5" single-density )
 5  17 c, 17 c,  34 c, 17 c, ( 5" double-sensity FHL FLEX )
 6  18 c, 18 c,  36 c, 18 c, ( 5" double-density )
 7  15 c, 15 c,  30 c, 15 c, ( 8" single-density )
 8  26 c, 26 c,  52 c, 26 c, ( 8" double-density )
 9  29 c, 29 c,  58 c, 29 c, ( 8" SWTP extra-density )
10   0 ,  ( end of table sentinel )
11 here SectorCounts - 2- 2/ constant Entries ( -- size )
12 : SetSides ( sectors -- )  SectorCounts
13    Entries 0 do 2dup c@ = ?leave 2+ loop
14    1+ c@ ?dup 0= abort" Unrecognizable format."
15    s/s c!   sectors c! ;
```

**Block # 52**

```
 0 ( ClearDisk Remove BackUp                   12:47pm cee 23jan84 )
 1 : ClearDisk ( -- )  pad b/blk blank
 2    pad Bounds over + swap ?do i false r/w loop drop ;
 3
 4 : Remove ( dr# -- )  >Drive 0 blocks ! ;
 5
 6 : BackUp ( FromDr# ToDr# -- )
 7    swap >Drive Bounds rot >Drive Bounds min editor copies ;
 8
 9 : Restore ( -- ) origin 10 + @ execute ;
10
11 : ReadSector ( adr dadr -- )
12   4 0 do origin 6 + @ execute 0= ?leave Restore loop ?status ;
13
14 : WriteSector ( adr dadr -- )
15   4 0 do origin 8 + @ execute 0= ?leave Restore loop ?status ;
```

**Block # 53**

```
 0 ( Claim  Release  Mount                     12:47pm cee 23jan84 )
 1
 2 : Claim ( cnt -- ) Configure sectors c@ SetSides
 3     dup blocks ! s/b c@ * sectors c@ /mod 'claim @ execute
 4     Bounds drop dup scr ! block dup c/l blank
 5     10272 over ! 2 r# ! editor >i insert
 6     Drive @ swap 1008 + Size cmove ;
 7
 8 : Release ( cnt -- )
 9     Configure tracks c@ sectors c@ * swap - Claim ;
10
11 : Mount ( dr# -- )
12     >Drive Bounds drop block
13     dup @ 10272 - abort" Unclaimed Disk."
14     1008 + Drive @ Size cmove ;
15
```

Block # 54
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Block # 55
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

Block # 56
```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 57**

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

**Block # 58**

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

**Block # 59**

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

**Block # 60**

```
 0 cr .(      HARDWARE DEPENDENT OPTIONS      12:47pm cee 23jan84 )
 1
 2 ( Remove the "|" from lines which apply to your system. )
 3 |    1 +load | 132 column printer such as Epson MX80
 4 |    2 +load | cursor control - eFORTH/CoCo
 5 |    3 +load | cursor control - FHL FLEX
 6 |    4 +load | cursor control - TeleVideo
 7 |    5 +load | cursor control - template
 8 exit
 9 The other blocks contain alternate definitions of date and
10 time which take advantage of various hardware capabilities.
11
12 If you have FLEX then block 78 should replace block 4.
13
14 If you have a Gimix CPU board, then 78 should replace 4 and
15
```

**Block # 61**

```
 0 ( index  listing  show    6/page            12:47pm cee 23jan84 )
 1
 2 : index ( n1 n2 --- )
 3    swap  dup 60 mod if header then
 4    do i 60 mod if cr else header then
 5       i block i 5 .r space c/l -trailing  type loop cr ;
 6 : list2 ( blk -- )  scr ! cr ." Block # " scr @ 4 .r
 7     54 spaces ." Block " scr @ 1+ 4 .r   b/blk c/l / 0
 8     do cr i 2 .r space scr @ block i c/l * + c/l type space
 9       scr @ 1+ block i c/l * + c/l type loop cr ;
10 : listing ( scr -- )  header
11     6 / 6 * dup 6 + swap do cr i list2 2 +loop ;
12 : show ( beg end -- )  swap do i listing 6 +loop ;
13
14 system  printer output 132 'width ! term output forth
15
```

**Block # 62**

```
 0 ( cursor control - eFORTH/CoCo                12:47pm cee 23jan84 )
 1 ( These versions are for the Color Computer version of eFORTH. )
 2 system definitions
 3 : (page) ( -- ) 26 emit ;
 4 : (xy)   ( x y -- ) 20 emit 32 + emit 32 + emit ;
 5 : (home) ( -- ) 30 emit ;
 6 : (eol)  ( -- )  5 emit ;
 7 : (eos)  ( -- ) 19 emit ;
 8
 9 term output          ' (page) 'page !
10                      ' (xy) 'xy !
11                      ' (home) 'home !
12                      ' (eos) 'eos !
13                      ' (eol) 'eol !
14
15
```

**Block # 63**

```
 0 ( cursor control - FHL FLEX                       12:47pm cee 23jan84 )
 1 ( These versions are for FHL Color Computer FLEX )
 2 system definitions
 3 : (page) ( -- ) 2 emit ;
 4 : (xy)   ( x y -- ) 20 emit 32 + emit 32 + emit ;
 5 : (home) ( -- ) 15 emit ;
 6 : (eol)  ( -- )  5 emit ;
 7 : (eos)  ( -- ) 19 emit ;
 8
 9 term output           ' (page) 'page !
10                       ' (xy) 'xy !
11                       ' (home) 'home !
12                       ' (eos) 'eos !
13                       ' (eol) 'eol !
14
15 forth definitions
```

**Block # 64**

```
 0 ( cursor control - TeleVideo                      12:47pm cee 23jan84 )
 1 ( These versions are for TeleVideo terminals )
 2 system definitions
 3 : (page) ( -- ) 26 emit ;
 4 : (xy)   ( x y -- ) 27 emit ascii = emit 32 + emit 32 + emit ;
 5 : (home) ( -- ) 30 emit ;
 6 : (eol)  ( -- ) 27 emit ascii T emit ;
 7 : (eos)  ( -- ) 27 emit ascii Y emit ;
 8
 9 term output           ' (page) 'page !
10                       ' (xy) 'xy !
11                       ' (home) 'home !
12                       ' (eos) 'eos !
13                       ' (eol) 'eol !
14
15 forth definitions
```

**Block # 65**

```
 0 ( cursor control - template                       12:47pm cee 23jan84 )
 1 ( This block is a form for defining these for other terminals. )
 2 system definitions
 3 : (page) ( -- )      ;
 4 : (xy)   ( x y -- )       ;
 5 : (home) ( -- )      ;
 6 : (eol)  ( -- )      ;
 7 : (eos)  ( -- )      ;
 8
 9 term output           ' (page) 'page !
10                       ' (xy) 'xy !
11                       ' (home) 'home !
12                       ' (eos) 'eos !
13                       ' (eol) 'eol !
14
15 forth definitions
```

**Block # 66**

```
 0 ( date - FLEX                                  12:47pm cee 23jan84 )
 1
 2 hex
 3
 4 : date ( -- adr cnt ) ( uses FLEX date registers )
 5    <# CC10 c@ 0  # # 2drop  CC0E c@
 6    1- 3 * " janfebmaraprmayjunjulaugsepoctnovdec" drop
 7    + 0 2 do dup i + c@ hold -1 +loop drop
 8    CC0F c@ 0 # #  #> ;
 9
10 decimal
11
12
13
14
15
```

**Block # 67**

```
 0 ( date - Gimix CPU board                       12:47pm cee 23jan84 )
 1
 2 hex  84 constant year
 3
 4 : date ( -- adr cnt )  base @ hex
 5    year 0 <#  # # 2drop  E227 c@ dup 9 > if 6 - then
 6    1- 3 * " janfebmaraprmayjunjulaugsepoctnovdec" drop
 7    + 0 2 do dup i + c@ hold -1 +loop drop
 8    E226 c@ 0 # #  #> rot base ! ;
 9
10 decimal
11
12
13
14
15
```

**Block # 68**

```
 0 ( time - Gimix CPU board                       12:47pm cee 23jan84 )
 1
 2 hex
 3 : time ( -- adr cnt )
 4    base @ hex <#  ascii m hold   E224 c@ 11 >
 5    if ascii p hold else ascii a hold then
 6    E223 c@ 0 # #  ascii : hold 2drop
 7    E224 c@ dup 1 < if 12 + else dup 12 > if dup 20 22 within
 8    if 18 - else 12 - then then then
 9    0 # #   #> rot base ! ;
10
11 decimal
12
13
14
15
```

**Block # 69**

```
 0 ( reserved                                  12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 70**

```
 0 ( reserved                                  12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

**Block # 71**

```
 0 ( reserved                                  12:47pm cee 23jan84 )
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

# APPENDIX D

## eFORTH INSTALLATION - FLEX

### REQUIREMENTS

The FLEX implementation of eFORTH (eFORTH/FLEX) requires the FLEX operating system and at least 32K of RAM (at least 40K is recommended). No special hardware is required.

### MAKE A BACKUP!

eFORTH/FLEX is distributed on either single-density, single-sided 8" disks or double-density, single-sided 5" disks. The following instructions assume that you have received a disk from us in one of these formats.

1. Using FLEX, format one disk for each drive that you have. You may use any format that works on your drives. We will call these disks "your" disks. One will be "your drive 0" disk, the second will be "your drive 1" disk, etc. We will assume that your system drive is drive 0.

If you are using FHL FLEX for the Color Computer, follow the directions in Appendix E for making a backup.

2. Write-protect the supplied disk with eFORTH on it by covering the notch on the disk (5" disks) or uncovering it (8" disks). We'll call this "our" disk.

3. Put "our" disk in drive 0 and enter EFORTH.CMD and hit return.

4. Put "your" drive 1 disk into drive 1, etc. Set "your" drive 0 disk aside for the moment.

5. When eFORTH starts running, enter

```
system disking ( You must use lower case. )
1 >Drive       ( The 'D' must be upper case. )
0 Release      ( The 'R' must be upper case. )
2 >Drive       ( Only if you have three drives. )
0 Release      ( Only if you have three drives. )
3 >Drive       ( Only if you have four drives. )
0 Release      ( Only if you have four drives. )
```

6. Now remove "your" drive 1 disk from drive 1 and put "your" drive 0 disk into drive 1 (yes, drive 1). Enter

```
      32 Release      ( The 'R' must be upper case. )
       0 1 BackUp     ( Both 'B' and 'U' must be upper case.)
```

7. Remove "our" disk from drive 0 and replace it with your FLEX system disk. eFORTH should still be running. Enter

**here hex u.**

and hit return. Remember the number that's printed. Let's suppose it's 4CD0. Now enter

**" save,1.forth.cmd,0,4CD0,0" dos**

(be sure a space follows both quotation marks) and wait until FLEX is done creating a FORTH.CMD file on "your" drive 0 disk.

8. Put "our" disk away in a nice, safe place, and don't use it again unless something terrible happens to "your" disk. In that case, use "our" disk to make another "your" disk.

9. Remove your FLEX system disk from drive 0 and replace it with "your" drive 0 disk, then put "your" drive 1 disk back into drive 1.

10. Go FORTH!

## RUNNING eFORTH

After you have performed the above installation process, eFORTH is run by simply putting "your" drive 0 disk into drive 0, "your" drive 1 disk into drive 1, etc. and entering FORTH (from FLEX).

## eFORTH DISK ACCESS

If you followed the above procedure, "your" drive 0 disk is "partitioned". Part of it is used by FORTH, and FLEX doesn't know about that part. Part of it is used by FLEX, and FORTH doesn't know about that part.

The phrase **0 Release** reserves the entire disk for FORTH. The phrase **32 Release** releases 32K bytes on the disk for the use of FLEX. Similarly, the phrase **32 Claim** will claim 32K bytes of the disk for FORTH, the rest will be left for FLEX. **Claim** and

 **copyright 1983**

Release will only work on a freshly formatted disk.

### CHANGING DISKS

If you change the disk in a drive and the new disk has a different format or has a different number of blocks claimed or released then you must "mount" it with **Mount** which must be preceded with the drive number. For example,

**1 Mount**

will mount a new disk in drive 1.

In order for **Mount** to work correctly, the disk must have been "claimed" with either **Claim** or **Release** .

### CALLING FLEX FROM FORTH

The above procedure uses the word **dos** which is used to pass a string to FLEX to be interpreted as a FLEX command. Be careful with it. Some FLEX commands, such as COPY.CMD and NEWDISK.CMD will destroy eFORTH. Commands such as SAVE.CMD, CAT.CMD, and LIST.CMD which only use the utility command space work just fine. FLEX will report any disk errors that arise, but control will return to eFORTH.

The source code for FLEX specific words will be found on blocks 72 through 83.

### THE .COR FILE

If you decide to change some of the words which appear on blocks 1 through 83, then , after you have used the editor to make your changes, Execute the EFORTH.COR file. When eFORTH starts, enter **1 load** and prepare for a wait. When eFORTH finally says "ok", you may use the "save" procedure described above to create a new .CMD file which has all of your changes in it.

# APPENDIX E

## eFORTH INSTALLATION - COCO

### REQUIREMENTS

The TRS-80 Color Computer implementation of eFORTH (eFORTH/COCO) requires at least one disk drive and Disk Extended BASIC. It also requires 64K of RAM. It will not work in 16K or 32K Color Computers.

### MAKE A BACKUP!

eFORTH/COCO is distributed on double-density, single-sided 5" diskettes. The following instructions assume that you have received a disk from us in this format.

1. Write-protect the supplied disk with eFORTH on it by covering the notch on the disk. We'll call this "our" disk.

2. While in BASIC use the BACKUP command to copy "our" disk onto another empty, freshly formatted disk. We'll call this "your" disk.

3. Put "our" disk away in a nice, safe place, and don't use it again unless something terrible happens to "your" disk. In that case, use "our" disk to make another "your" disk.

4. Now put "your" disk in drive 0 and enter

   **LOADM"EFORTH"**

and hit the enter key.

5. When BASIC says "OK", enter EXEC and hit the enter key. eFORTH will sign on and wait for you to give it something to do.

6. Go FORTH!

If you have another disk drive (drive 1), place an empty, freshly formatted disk in it and enter

```
system disking      ( You must use lower case. )
1 >Drive            ( The 'D' must be upper case. )
0 Release           ( The 'R' must be upper case. )
```

**eFORTH DISK ACCESS**

If you followed the above procedure, "your" drive 0 disk is "partitioned". Part of it is used by FORTH, and BASIC doesn't know about that part. Part of it is used by BASIC, and FORTH doesn't know about that part.

The phrase **0 Release** reserves the entire disk for FORTH. The phrase **32 Release** releases 32K bytes on the disk for the use of BASIC. Similarly, the phrase **32 Claim** will claim 32K bytes of the disk for FORTH, the rest will be left for BASIC. **Claim** and **Release** will only work on a freshly formatted disk.

**CHANGING DISKS**

If you change the disk in a drive and the new disk has a different format or has a different number of blocks claimed or released, then you must "mount" it with **Mount** which must be preceded with the drive number. For example,

**1 Mount**

will mount a new disk in drive 1.

In order for **Mount** to work correctly, the disk must have been "claimed" with either **Claim** or **Release** .

If you define new words and want them to be available whenever you LOADM"FORTH", then do the following:

First enter **hex here u.** and write down the number that is printed. Let's suppose that it's 3AB7. Now enter **system mon** and you will be back in BASIC. Now enter

**SAVEM"FORTH",&H1A00,&H3AB7,&H1A00**

and hit the enter key. If there is enough room on the disk, the file FORTH/BIN will be created. Now, whenever you run eFORTH, all of the words will be in your dictionary that were there when

you saved it.

The source code for Color Computer specific words will be found on blocks 72 through 83.

## THE /COR FILE

If you decide to change some of the words which appear on blocks 1 through 83, then , after you have used the editor to make your changes, EXEC the EFORTH/COR file. When eFORTH starts, enter **1 load** and prepare for a wait. When eFORTH finally says "ok", you may use the SAVEM procedure described above to create a new /BIN file which has all of your changes in it.

## eFORTH KEYBOARD INTERPRETATION

eFORTH interprets the keyboard differently than BASIC. The following chart shows the ASCII code that each key returns to eFORTH. the "SHIFT" column means that the SHIFT key is held down at the same time. The "CONTROL" column means that the CLEAR key is held down at the same time. So, "control-X" means to hold down the CLEAR key, then press the "X" key, then let up on both of them. The codes are given in hexadecimal (base 16).

| | NORM | SHIFT | CONTROL |
|---|---|---|---|
| BREAK | 1B | 1B | 1B |
| ENTER | 0D | 0D | 0D |
| SPACE | 20 | 20 | 20 |
| < | 08 | 18 | 10 |
| > | 09 | 19 | 11 |
| v | 0A | 1A | 12 |
| ^ | 0B | 1B | 13 |

| NORM | SHIFT | CONTROL |
|---|---|---|
| 0 30 | 0 30 | *toggle* |
| 1 31 | ! 21 | \| 7C |
| 2 32 | " 22 | 00 |
| 3 33 | # 23 | ~ 7E |
| 4 34 | $ 24 | 00 |
| 5 35 | 4 25 | 00 |
| 6 36 | & 26 | 00 |
| 7 37 | ' 27 | ^ 5E |
| 8 38 | ( 28 | [ 5B |
| 9 39 | ) 29 | ] 5D |
| ; 3B | + 2B | 00 |
| , 2C | < 3C | { 7B |
| - 2D | = 3D | _ 5F |
| / 2F | ? 3F | \ 5C |

*toggle* means that this works the same way it does in BASIC.

 copyright 1983

| NORM | SHIFT | CONTROL |
|---|---|---|
| @ 40 | ` 60 | 00 |
| A 41 | a 61 | 01 |
| B 42 | b 62 | 02 |
| C 43 | c 63 | 03 |
| D 44 | d 64 | 04 |
| E 45 | e 65 | 05 |
| F 46 | f 66 | 06 |
| G 47 | g 67 | 07 |
| H 48 | h 68 | 08 |
| I 49 | i 69 | 09 |
| J 4A | j 6A | 0A |
| K 4B | k 6B | 0B |
| L 4C | l 6C | 0C |
| M 4D | m 6D | 0D |
| N 4E | n 6E | 0E |
| O 4F | o 6F | 0F |
| P 50 | p 70 | 10 |
| Q 51 | q 71 | 11 |
| R 52 | r 72 | 12 |
| S 53 | s 73 | 13 |
| T 54 | s 74 | 14 |
| U 55 | u 75 | 15 |
| V 56 | v 76 | 16 |
| W 57 | w 77 | 17 |
| X 58 | x 78 | 18 |
| Y 59 | y 79 | 19 |
| Z 5A | z 7A | 1A |

## THE eFORTH/COCO DISPLAY

The video display uses a high-resolution graphics mode to produce a display format of 24 lines with 51 characters on each line. It is quite readable on most TV sets.

The display can be controlled by emitting control characters. The available operations are:

**1 emit ( toggle the cursor from underline to block and back )**
**2 emit ( toggle the cursor from steady to blinking and back )**
**5 emit ( erase from the cursor to the end of the line )**
**7 emit ( ring the bell )**
**8 emit ( move the cursor to the left )**
**9 emit ( move the cursor to the right )**
**10 emit ( move the cursor down one line )**
**11 emit ( move the cursor up one line )**
**13 emit ( move the cursor to the left margin )**
**15 emit ( move the cursor to the upper left corner )**
**19 emit ( erase from the cursor to the end of the screen )**
**20 emit ( move the cursor to the specified location )**
**23 emit ( insert line )**
**24 emit ( delete line )**
**26 emit ( home cursor and erase the screen )**

The "insert line" function moves the current line and all lines below it down one line. The bottom line is lost. The "delete line" function moves the current line and all lines above it up one line. The top line is lost. The "move cursor" function requires the line number and the column number on that line to be specified. For example,

**20 emit 32 emit 32 emit**

will move the cursor to the upper left corner (the "home" function), and

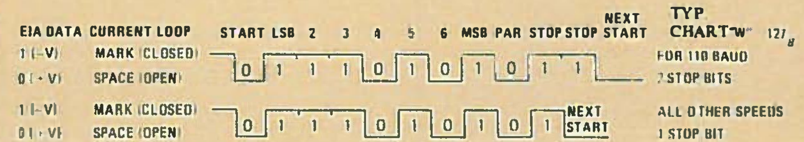
**20 emit 33 emit 32 emit**

will move the cursor to column 0 on line 1. Notice that 32 must be added to the column and line number.

 copyright 1983

# The ASCII Code

BASED ON ANSI X3.4 1968




| Octal | Dec. | Hex | Character | Control Keybd. Equiv. | Alternate Code Names |
|---|---|---|---|---|---|
| 000 | 0 | 00 | NUL | @ | NULL, CTRL SHIFT P, TAPE LEADER |
| 001 | 1 | 01 | SOH | A | START OF HEADER, SOM |
| 002 | 2 | 02 | STX | B | START OF TEXT, EOA |
| 003 | 3 | 03 | ETX | C | END OF TEXT, EOM |
| 004 | 4 | 04 | EOT | D | END OF TRANSMISSION, END |
| 005 | 5 | 05 | ENO | E | ENQUIRY, WRU, WHO ARE YOU |
| 006 | 6 | 06 | ACK | F | ACKNOWLEDGE, RU, ARE YOU |
| 007 | 7 | 07 | BEL | G | BELL |
| 010 | 8 | 08 | BS | H | BACKSPACE, FE0 |
| 011 | 9 | 09 | HT | I | HORIZONTAL TAB, TAB |
| 012 | 10 | 0A | LF | J | LINE FEED, NEW LINE, NL |
| 013 | 11 | 08 | VT | K | VERTICAL TAB, VTAB |
| 014 | 12 | 0C | FF | L | FORM FEED, FORM, PAGE |
| 015 | 13 | 0D | CR | M | CARRIAGE RETURN, EOL |
| 016 | 14 | 0E | SO | N | SHIFT OUT, RED SHIFT |
| 017 | 15 | 0F | SI | O | SHIFT IN, BLACK SHIFT |
| 020 | 16 | 10 | DLE | P | DATA LINK ESCAPE, DC0 |
| 021 | 17 | 11 | DC1 | Q | XON, READER ON |
| 022 | 18 | 12 | DC2 | R | TAPE, PUNCH ON |
| 023 | 19 | 13 | DC3 | S | XOFF, READER OFF |
| 024 | 20 | 14 | DC4 | T | TAPE, PUNCH OFF |
| 025 | 21 | 15 | NAK | U | NEGATIVE ACKNOWLEDGE, ERR |
| 026 | 22 | 16 | SYN | V | SYNCHRONOUS IDLE, SYNC |
| 027 | 23 | 17 | ETB | W | END OF TEXT BUFFER, LEM |
| 030 | 24 | 18 | CAN | X | CANCEL, CANCL |
| 031 | 25 | 19 | EM | Y | END OF MEDIUM |
| 032 | 26 | 1A | SUB | Z | SUBSTITUTE |
| 033 | 27 | 1B | ESC | [ | ESCAPE, PREFIX |
| 034 | 28 | 1C | FS | \ | FILE SEPARATOR |
| 035 | 29 | 1D | GS | ] | GROUP SEPARATOR |
| 036 | 30 | 1E | RS | ^ | RECORD SEPARATOR |
| 037 | 31 | 1F | US | _ | UNIT SEPARATOR |

| Octal | Dec. | Hex | Character | Alternates |
|---|---|---|---|---|
| 040 | 32 | 20 | SP | SPACE, BLANK |
| 041 | 33 | 21 | ! | |
| 042 | 34 | 22 | " | |
| 043 | 35 | 23 | # | |
| 044 | 36 | 24 | $ | |
| 045 | 37 | 25 | % | |
| 046 | 38 | 26 | & | |
| 047 | 39 | 27 | ' | APOSTROPHE |
| 050 | 40 | 28 | ( | |
| 051 | 41 | 29 | ) | |
| 052 | 42 | 2A | * | |
| 053 | 43 | 2B | + | |
| 054 | 44 | 2C | , | COMMA |
| 055 | 45 | 2D | - | MINUS |
| 056 | 46 | 2E | . | |
| 057 | 47 | 2F | / | |
| 060 | 48 | 30 | 0 | NUMBER ZERO |
| 061 | 49 | 31 | 1 | NUMBER ONE |
| 062 | 50 | 32 | 2 | |
| 063 | 51 | 33 | 3 | |
| 064 | 52 | 34 | 4 | |
| 065 | 53 | 35 | 5 | |
| 066 | 54 | 36 | 6 | |
| 067 | 55 | 37 | 7 | |
| 070 | 56 | 38 | 8 | |
| 071 | 57 | 39 | 9 | |
| 072 | 58 | 3A | : | |
| 073 | 59 | 3B | ; | |
| 074 | 60 | 3C | < | LESS THAN |
| 075 | 61 | 3D | = | |
| 076 | 62 | 3E | > | GREATER THAN |
| 077 | 63 | 3F | ? | |

| Octal | Dec. | Hex | Character | Alternates |
|---|---|---|---|---|
| 100 | 64 | 40 | @ | |
| 101 | 65 | 41 | A | |
| 102 | 66 | 42 | B | |
| 103 | 67 | 43 | C | |
| 104 | 68 | 44 | D | |
| 105 | 69 | 45 | E | |
| 106 | 70 | 46 | F | |
| 107 | 71 | 47 | G | |
| 110 | 72 | 48 | H | |
| 111 | 73 | 49 | I | LETTER I |
| 112 | 74 | 4A | J | |
| 113 | 75 | 4B | K | |
| 114 | 76 | 4C | L | |
| 115 | 77 | 4D | M | |
| 116 | 78 | 4E | N | |
| 117 | 79 | 4F | O | LETTER O |
| 120 | 80 | 50 | P | |
| 121 | 81 | 51 | Q | |
| 122 | 82 | 52 | R | |
| 123 | 83 | 53 | S | |
| 124 | 84 | 54 | T | |
| 125 | 85 | 55 | U | |
| 126 | 86 | 56 | V | |
| 127 | 87 | 57 | W | |
| 130 | 88 | 58 | X | |
| 131 | 89 | 59 | Y | |
| 132 | 90 | 5A | Z | |
| 133 | 91 | 5B | [ | |
| 134 | 92 | 5C | \ | |
| 135 | 93 | 5D | ] | |
| 136 | 94 | 5E | ^ | |
| 137 | 95 | 5F | _ | UNDERSCORE |

| Octal | Dec. | Hex | Character | Alternates |
|---|---|---|---|---|
| 140 | 96 | 60 | ` | ACCENT GRAVE |
| 141 | 97 | 61 | a | |
| 142 | 98 | 62 | b | |
| 143 | 99 | 63 | c | |
| 144 | 100 | 64 | d | |
| 145 | 101 | 65 | e | |
| 146 | 102 | 66 | f | |
| 147 | 103 | 67 | g | |
| 150 | 104 | 68 | h | |
| 151 | 105 | 69 | i | |
| 152 | 106 | 6A | j | |
| 153 | 107 | 68 | k | |
| 154 | 108 | 6C | l | |
| 155 | 109 | 6D | m | |
| 156 | 110 | 6E | n | |
| 157 | 111 | 6F | o | |
| 160 | 112 | 70 | p | |
| 161 | 113 | 71 | q | |
| 162 | 114 | 72 | r | |
| 163 | 115 | 73 | s | |
| 164 | 116 | 74 | t | |
| 165 | 117 | 75 | u | |
| 166 | 118 | 76 | v | |
| 167 | 119 | 77 | w | |
| 170 | 120 | 78 | x | |
| 171 | 121 | 79 | y | |
| 172 | 122 | 7A | z | |
| 173 | 123 | 78 | { | |
| 174 | 124 | 7C | \| | VERTICAL SLASH |
| 175 | 125 | 7D | } | |
| 176 | 126 | 7E | ~ | |
| 177 | 127 | 7F | DEL | |

To transmit any control code (first column), depress "CTRL" then the character on the same line under Keyboard Equivalent


FHL FRANK HOGG LABORATORY

THE REGENCY TOWER • 770 JAMES ST • SYRACUSE NY 13203 • (315) 474 7856