Programming in FORTH

Contents

This chapter briefly reviews the main ideas of FORTH to let the reader understand the program fragments and subroutines that comprise the meat of this book. We make no pretense to complete coverage of standard FORTH programming methods. **Chapter 2 is not a programmer's manual!**

Suppose the reader is stimulated to try FORTH – how can he proceed? Several excellent FORTH texts and references are available: *Starting FORTH*[1] and *Thinking FORTH*[2] by Leo Brodie; and *FORTH: a Text and Reference*[3] by M.Kelly and N.Spies. I strongly recommend reading **FTR** or **SF** (or both) before trying to use the ideas from this book on a FORTH system. (Or at least read one concurrently.)

The (commercial) GEnie information network maintains a session devoted to FORTH under the aegis of the Forth Interest Group (FIG).

FIG publishes a journal *Forth Dimensions* whose object is the exchange of programming ideas and clever tricks.

The Association for Computing Machinery (11 West 42nd St., New York, NY 10036) maintains a Special Interest Group on FORTH (SIGForth).

The Institute for Applied FORTH Research (Rochester, NY) publishes the refereed *Journal of FORTH Application and Research*, that serves as a vehicle for more scholarly and theoretical papers dealing with FORTH.

Finally, an attempt to codify and standardize FORTH is underway, so by the time this book appears the first draft of an ANS FORTH and extensions may exist.

---

1. L. Brodie, *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986), referred to hereafter as **SF**.
2. L. Brodie, *Thinking FORTH* (Prentice-Hall, NJ 1984), referred to hereafter as **TF**.
3. M. Kelly and N. Spies, *FORTH: a Text and Reference* (Prentice-Hall, NJ, 1986), referred to hereafter as **FTR**.

§1 The structure of FORTH

The "atom" of FORTH is a **word** – a previously-defined operation (defined in terms of machine code or other, previously-defined words) whose definition is stored in a series of linked lists called the **dictionary**. The FORTH operating system is an endless loop (outer interpreter) that reads the console and interprets the input stream, consulting the dictionary as necessary. If the stream contains a word[4] in the dictionary the interpreter immediately executes that word.

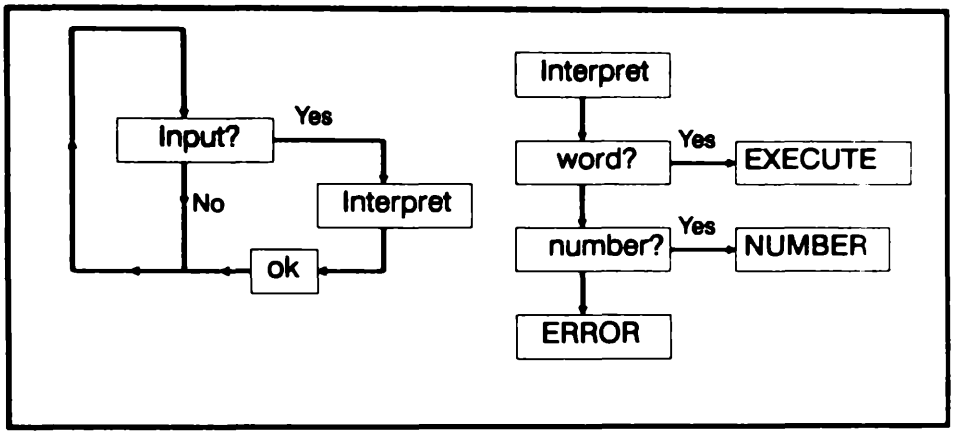

Fig. 2-1 *Overview of FORTH outer interpreter*

In general, because FORTH is interpretive as well as compiled, the best way to study something new is in front of a computer running FORTH. Therefore we explain with illustrations, expecting the reader to try them out.

In what follows, anything the user types in will be set in Helvetica, such as DECIMAL below.

Machine responses appear in ordinary type.

We now give a trivial illustration:

```
DECIMAL  <cr> ok
```

4. Successive words in the input stream are separated from each other by blank spaces, ASCII 20hex, the standard FORTH delimiter.

Notes:

- **<cr>** means "the user pushes the ENTER or ↵ button".
- **ok** is what FORTH says in response to an input line, if nothing has gone wrong.
- DECIMAL is an instruction to use base 10 arithmetic. FORTH will use any base you tell it, within reason, but usually only DECIMAL and HEX (hexadecimal) are predefined.

When the outer interpreter (see Fig. 2.1 on p. 13) encounters text with no dictionary entry, it tries to interpret it as a **NUMBER**.

It places the number in a special memory location called "the top of the stack" (TOS)[5]

```
2 17 + . <cr> 19 ok
```

Notes:

- FORTH interprets 2 and 17 as numbers, and pushes them onto the stack. "**+**" is a word and so is "**.**" so they are **EXECUTE**d.
- **+** adds 2 to 17 and leaves 19 on the stack.
- The word **.** (called "emit") removes 19 from the stack and displays it on the screen.

We might also have said[6]

```
HEX 0A 14 * . <cr> C8 ok
```

(Do you understand this? Hint: **HEX** stands for "switch to hexadecimal arithmetic".)

---

5. We will explain about the stack in §2.3.
6. since FORTH uses **words**, when we enter an input line we **say** the corresponding phrase.

If the incoming text can neither be located in the dictionary nor interpreted as a number, FORTH issues an error message.

§2 Extending the dictionary

The compiler is one of FORTH's most endearing features. It is elegant, simple, and mostly written in FORTH. Although the technical details of the FORTH compiler are generally more interesting to systems developers than to scientists, its components can often be used to solve programming problems. When this is the case, we necessarily discuss details of the compiler. In this section we discuss how the compiler extends the dictionary. In §2§§8 below we examine the parts of the compiler in greater detail.

FORTH has special words that allow the creation of new dictionary entries, *i.e.*, new words. The most important are " : " ("start a new definition") and " ; " ("end the new definition").

Consider the phrase

```
: NEW-WORD WORD1  17  WORD2 . . . WORDn ; ok
```

The initial " : " is **EXECUTE**d because it is already in the dictionary. Upon execution, " : " does the following:

- Creates a new dictionary entry, **NEW-WORD**, and switches from **interpret**- to **compile** mode.
- In compile mode, the interpreter looks up words and – rather than executing them – installs pointers to their code. If the text is a number (17 above), FORTH builds the literal number into the dictionary space allotted for **NEW-WORD**.
- The action of **NEW-WORD** will be to **EXECUTE** sequentially the previously-defined words **WORD1**, **WORD2**, ...**WORDn**, placing any built-in numbers on the stack as they occur.

- The FORTH compiler **EXECUTE**s the last word " ; " of the definition, by installing code (to return control to the next outer level of the interpreter[7]) then switching back from compile to interpret mode. Most other languages treat tokens like " ; " as flags (in the input stream) that *trigger* actions, rather than actions in their own right. FORTH lets components execute themselves.

In FORTH *all* subroutines are words that are invoked when they are named. No explicit CALL or GOSUB statement is required.

The above definition of **NEW-WORD** is extremely structured compared with FORTRAN or BASIC. Its definition is just a series of subroutine calls.

We now illustrate how to define and use a new word using the previously defined words " : " and " ; ". Enter the phrase (this new word *+ expects 3 numbers, *a*, *b*, and *c* on the stack)

```
: *+   * + ; ok
```

Notes:

- * multiplies b with c, leaving b*c.
- + then adds b*c to a, leaving a + b*c behind.

Now we actually try out *+ :

```
DECIMAL 5 6 7 *+ . 47 ok
```

Notes:

- The period . is not a typo, it EMITs the result.
- FORTH's response to **a b c *+ .** is a + b*c ok .

7. This level could be either the outer interpreter or a word that invokes **NEW-WORD** .

What if we were to enter * + with nothing on the stack ? Let's try it and see (.S is a word that displays the stack without changing its contents):

```
.S empty stack ok

* + empty stack ok
```

Exercise:

Suppose you entered the input line

```
HEX  5  6  7  * + .  <cr>  xxx  ok
```

What would you expect the response xxx to be?

Answer: 2F

---

§3 Stacks and reverse Polish notation (RPN)

We now discuss the stack and the "reverse Polish" or "postfix" arithmetic based on it. (Anyone who has used one of the Hewlett-Packard calculators should already be familiar with the basic concepts.)

A Polish mathematician (J.Lukasewcleicz) showed that numerical calculations require an irreducible minimum of elementary operations (fetching and storing numbers as well as addition, subtraction, multiplication and division). The minimum is obtained when the calculation is organized by "stack" arithmetic.

Thus virtually all central processors (CPU's) intended for arithmetic operations are designed around stacks. FORTH makes efficient use of CPU's by reflecting this underlying stack architecture in its syntax, rather than translating algebraic-looking program statements ("infix" notation) into RPN-based machine operations as FORTRAN, BASIC, C and Pascal do.

But what *is* a stack? As the name implies, a stack is the machine analog of a pile of cards with numbers written on them. Numbers are always added to, and removed from, the top of the pile. (That is, a stack resembles a job where layoffs follow seniority: last in, first out.) Thus, the FORTH input line

```
DECIMAL 2 5 73 -16 ok
```

followed by the line

```
+ - * . yyy ok
```

leaves the stack in the successive states shown in Table 2-1 below

Cell #	Initial	Ops→	+	–	*	.
0	-16	Result	57	-52	-104	...
1	73	→	5	2	...	...
2	5		2	...	...	
3	2		...'	...		

Table 2-1 *Picture of the stack during operations*

We usually employ zero-based relative numbering in FORTH data structures —stacks, arrays, tables, *etc.*— so TOS ("top of stack") is given relative #0, NOS ("next on stack") #1, *etc.*

The operation "." ("emit") displays -104 to the screen, leaving the stack empty. That is, **yyy** above is **-104**.

§1 Manipulating the parameter stack

FORTH systems incorporate (at least) two stacks: the **parameter stack** which we now discuss, and the **return stack** which we defer to §2.3.2.

In order to use a stack-based system, we must be able to put numbers on the stack, remove them, and rearrange their order. FORTH includes standard words for this purpose.

Putting numbers on the stack is easy: one simply types the number (or it appears in the definition of a FORTH word).

To remove a number we have the word **DROP** that drops the number from TOS and moves up all the other numbers.

To exchange the top 2 numbers we have .

DUP duplicates the TOS into NOS, pushing down all the other numbers.

ROT rotates the top 3 numbers.

Cell #	Initial	Ops→	DROP	SWAP	ROT	DUP
0	-16	Result	73	73	5	-16
1	73	→	5	-16	-16	-16
2	5		2	5	73	73
3	2		...	2	2	5
4	...		...	...	...	2

Table 2-2 *Stack manipulation operators*

These actions are shown on page 19 above in Table 2-2 (we show what each word does to the initial stack).

In addition the words **OVER, UNDER, PICK** and **ROLL** act as shown in Table 2-3 below (note **PICK** and **ROLL** must be

Cell #	Initial	Ops→	OVER	UNDER	4 PICK	4 ROLL
0	-16	Result	73	-16	2	
1	73	→	-16	73	-16	
2	5		73	-16	73	
3	2		5	5	5	
4	...		2	2	2	...

Table 2-3 *More stack manipulation operators*

preceded by an integer that says where on the stack an element gets **PICK**ed or **ROLL**ed).

Clearly, **1 PICK** is the same as **DUP**, **2 PICK** is a synonym for **OVER**, **2 ROLL** means **SWAP**, and **3 ROLL** means **ROT**.

As Brodie has noted (TF), it is rarely advisable to have a word use a stack so deep that **PICK** or **ROLL** is needed. It is generally better to keep word definitions short, using only a small number of arguments on the stack and consuming them to the extent possible. On the other hand, **ROT** and its opposite, **–ROT**[8], are often useful.

8. defined as : **–ROT ROT ROT ;**

§§2 The return stack and its uses

We have remarked above in §2§§2 that compilation establishes links from the calling word to the previously- defined word being invoked. Part of the linkage mechanism —during actual execution— is the **return stack** (rstack): the address of the next word to be invoked after the currently executing word is placed on the rstack, so that when the current word is done, the system jumps to the next word. Although it might seem logical to call the address on the rstack the **next** address, it is actually called the **return** address for historical reasons.

In addition to serving as a reservoir of return addresses (since words can be nested, the return addresses need a stack to be put on) the rstack is where the limits of a **DO ... LOOP** construct are placed[9].

The user can also store/retrieve to/from the rstack. This is an example of using a component for a purpose other than the one it was designed for. Such use is not encouraged by every FORTH text, needless to say, since it introduces the spice of danger. To store to the rstack we say **>R**, and to retrieve we say **R>** . **DUP >R** is a speedup of the phrase **DUP >R** . The words **D>R DR>** , for moving double-length integers, also exist on many systems. The word **R@** copies the top of the rstack to the TOS.

The danger is this: anything put on the rstack during a word's execution must be removed before the word terminates. If the **>R** and the **R>** do not balance, then a **wrong next address** will be jumped to and **EXECUTE**d. Since this could be the address of data, and since it is being interpreted as machine instructions, the results will be **always unpredictable**, but seldom amusing.

Why would we want to use the rstack for storage when we have a perfectly good parameter stack to play with? Sometimes it becomes simply impossible to read code that performs complex gymnastics on the parameter stack, even though FORTH permits such gymnastics.

---

9. We discuss looping in §2.7 below.

Consider a problem – say, drawing a line on a bit-mapped graphics output device from (x,y) to (x',y') – that requires 4 arguments. We have to turn on the appropriate pixels in the memory area representing the display, in the ranges from the origin to the end coordinates of the line. Suppose we want to work with x and y first, but they are 3rd and 4th on the stack. So we have to **ROLL** or **PICK** to get them to TOS where they can be worked with conveniently. We probably need them again, so we use

```
4 PICK 4 PICK  ( - - x y x' y' x y)
```

Now 6 arguments are on the stack! See what I mean? A better way stores temporarily the arguments x' and y', leaving only 2 on the stack. If we need to duplicate them, we can do it with an already existing word, **DDUP**.

Complex stack manipulations can be avoided by defining **VARIABLE**s – named locations – to store numbers. Since FORTH variables are typically *global* – any word can access them – their use can lead to unfortunate and unexpected interactions among parts of a large program. Variables should be used sparingly.

While FORTH permits us to make variables local to the subroutines that use them[10], for many purposes the rstack can advantageously replace local variables:

- The rstack already exists, so it need not be defined anew.
- When the numbers placed on it are removed, the rstack shrinks, thereby reclaiming some memory.

Suppose, in the previous example, we had put x' and y' on the rstack *via* the phrase

```
>R  >R  DDUP .
```

Then we could duplicate and access x and y with no trouble.

10. See **FTR**, p. 325ff for a description of beheading – a process to make variables local to a small set of subroutines. Another technique is to embed variables within a data structure so they cannot be referenced inadvertently. Chapters 2§8§§3-2, 3§5§§2, 5§1§§2 and 11§2 offer examples.

A note of caution: since the rstack is a critical component of the execution mechanism, we mess with it at our peril. If we want to use it, we must clean up when we are done, so it is in the same state as when we found it. A word that places a number on the rstack must get it off again – using **R>** or **RDROP** – before exiting that word[11]. Similarly, since **DO** ... **LOOP** uses the rstack also, for each **>R** in such a loop (after **DO**) there must be a corresponding **R>** or **RDROP** (before **LOOP** is reached). Otherwise the results will be unpredictable and probably will crash the system.

§4 Fetching and storing

Ordinary (16-bit) numbers are fetched from memory to the stack by " @ " ("fetch"), and stored by " ! " ("store"). The word @ expects an address on the stack and replaces that address by its contents using, *e.g.*, the phrase **X @** . The word " ! " expects a number (NOS) and an address (TOS) to store it in, and places the number in the memory location referred to by the address, consuming both arguments in the process, as in the phrase **32 X !**

Double length (32-bit) numbers can similarly be fetched and stored, by **D@** and **D!** . (FORTH systems designed for the newer 32-bit machines sometimes use a 32-bit-wide stack and may not distinguish between single- and double-length integers.)

Positive numbers smaller than 255 can be placed in single bytes of memory using **C@** and **C!** . This is convenient for operations with strings of ASCII text, for example screen, file and keyboard I/O.

In Chapters 3, 4, 5 and 7 we shall extend the lexicon of @ and ! words to include floating point and complex numbers.

---

11. **RDROP** is a handy way to exit from a word before reaching the final " ; " . See **TF**.

§5 Arithmetic operations

The 1979 or 1983 standards, not to mention the forthcoming ANSII standard, require that a conforming FORTH system contain a certain minimum set of predefined words. These consist of arithmetic operators **+ - * / MOD /MOD */** for (usually) 16-bit *signed-integer* (-32767 to +32767) arithmetic, and equivalents for *unsigned* (0 to 65535), double-length and mixed-mode (16- mixed with 32-bit) arithmetic. The list will be found in the glossary accompanying your system, as well as in **SF** and **FTR**.

§6 Comparing and testing

In addition to arithmetic, FORTH lets us **compare** numbers on the stack, using relational operators **> < =** . These operators work as follows: the phrase

```
2 3 > <cr> ok
```

will leave 0 ("false") on the stack, because 2 (NOS) is not greater than 3 (TOS). Conversely, the phrase

```
2 3 < <cr> ok
```

will leave -1 ("true") because 2 is less than 3. Relational operators typically consume their arguments and leave a "flag" to show what happened[12]. Those listed so far work with signed 16-bit integers. The operator **U<** tests *unsigned* 16-bit integers (0-65535).

FORTH offers unary relational operators **0= 0>** and **0<** that determine whether the TOS contains a (signed) 16-bit integer that is 0, positive or negative. Most FORTHs offer equivalent relational operators for use with double-length integers.

The relational words are used for branching and control. The usual form is

```
: MAYBE  0> IF WORD1 WORD2 ...
    WORDn THEN ;
```

---

12. The original FORTH-79 used +1 for "true", 0 for "false"; many newer systems that mostly follow FORTH-79 use -1 for "true". HS/FORTH is one such. Both FORTH-83 and ANSII FORTH require -1 for "true", 0 for "false".

The word **MAYBE** expects a number on the stack, and executes the words between **IF** and **THEN** if the number on the stack is positive, but not otherwise. If the number initially on the stack were negative or zero, **MAYBE** would do nothing.

An alternate form including **ELSE** allows two mutually exclusive actions:

```
: CHOOSE      0>  IF  WORD1 . . . WORDn
          ELSE WORD1' . . . WORDn'
          THEN ;       ( n - - )
```

If the number on the stack is positive, **CHOOSE** executes **WORD1 WORD2 ... WORD**, whereas if the number is negative or 0, **CHOOSE** executes **WORD1' ... WORDn'**.

In either example, **THEN** marks the end of the branch, rather than having its usual logical meaning[13].

§7 Looping and structured programming

FORTH contains words for setting up loops that can be definite or indefinite:

```
BEGIN xxx flag UNTIL
```

The words represented by **xxx** are executed, leaving the TOS (flag) set to 0 (F) – at which point **UNTIL** leaves the loop – or -1 (T) – at which point **UNTIL** makes the loop repeat from **BEGIN**.

A variant is

```
BEGIN xxx flag WHILE yyy REPEAT
```

Here **xxx** is executed, **WHILE** tests the flag and if it is 0 (F) leaves the loop; whereas if **flag** is -1 (T) **WHILE** executes **yyy** and

13. This has led some FORTH gurus to prefer the synonymous word **ENDIF** as clearer than **THEN**.

REPEAT then branches back to **BEGIN**. These forms can be used to set up loops that repeat until some external event (pressing a key at the keyboard, *e.g.*) sets the flag to exit the loop. They can also used to make endless loops (like the outer interpreter of FORTH) by forcing flag to be 0 in a definition like

```
: ENDLESS BEGIN xxx 0 UNTIL ;
```

FORTH also implements indexed loops using the words **DO LOOP +LOOP /LOOP**. These appear within definitions, *e.g.*

```
: LOOP-EXAMPLE 100 0 DO xxx LOOP ;
```

The words **xxx** will be executed 100 times as the lower limit, 0, increases in unit steps to 99. To step by –2's, we use the phrase

```
-2 +LOOP
```

to replace **LOOP**, as in

```
: DOWN-BY-2's 0 100 DO xxx -2 +LOOP ;
```

The word **/LOOP** is a variant of **+LOOP** for working with unsigned limits[14] and increments (to permit the loop index to go up to 65535 in 16-bit systems).

§8 The pearl of FORTH

An unusual construct, **CREATE ... DOES>**, has been called "the pearl of FORTH"[15]. This is more than poetic license.

CREATE is a component of the compiler that makes a new dictionary entry with a given name (the next name in the input stream) and has no other function.

DOES> assigns a specific run-time action to a newly **CREATE**d word (we shall see this in §2§§8-3 below).

---

14. Signed 16-bit integers run from –32768 to +32767, unsigned from 0 to 65535. See **FTR**.
15. Michael Ham, "Structured Programming", *Dr. Dobb's Journal of Software Tools*, October, 1986.

§§1 Dummy words

Sometimes we use **CREATE** to make a dummy entry that we can later assign to some action:

```
CREATE DUMMY
CA' * DEFINES DUMMY
```

The second line translates as "The code address of * defines **DUMMY**". Entry of the above phrase would let **DUMMY** perform the job of * just by saying **DUMMY**. That is, FORTH lets us first define a dummy word, and then give it any other word's meaning[16].

Here is one use of this power: Suppose we have to define two words that are alike except for some piece in the middle:

```
: *WORD  WORD1 WORD2  * WORD3 WORD4 ;
: */WORD WORD1 WORD2  */  WORD3  WORD4 ;
```

we could get away with 1 word, together with **DUMMY** from above,

```
: *_or_*/WORD
   WORD1  WORD2
   DUMMY
   WORD3  WORD4  ;
```

by saying

```
CA' * DEFINES  DUMMY  *_or_*/WORD
```

or

```
CA' */ DEFINES  DUMMY *_or_*/WORD .
```

---

16. This usage is a non-standard construct of HS/FORTH.