12 THREE EXAMPLES

Programming in FORTH is more of an "art" than programming in any other language. Like painters drawing brushstrokes, FORTH programmers have complete control over where they are going and how they will get there. Charles Moore has written, "A good programmer can do a fantastic job with FORTH; a bad programmer can do a disastrous job." A good FORTH programmer must be conscious of "style."

FORTH style is not easily taught; it's a subject that deserves a book of its own. Some elements of good FORTH style include:

simplicity,

the use of many short definitions rather than a few longer ones,

a correspondence between words and easy-to-understand actions or data structures,

well-chosen names, and

well laid-out blocks, clearly commented.

One good way to learn style, aside from trial and error, is to study existing FORTH applications, including FORTH itself. In this book we've included the definitions of many FORTH system words, and we encourage you to continue this study on your own.

This chapter introduces three applications which should serve as examples of good FORTH style.

The first example will show you the typical process of programming in FORTH: starting out with a problem and working step-by-step towards the solution.

The second example involves a more complex application already written: you will see the use of well-factored definitions and the creation of an application-specific "language."

The third example demonstrates the way to translate a mathematical equation into a FORTH definition; you will see how speed and compactness can be increased by using fixed-point arithmetic.

WORD Game

The example in this section is a refinement of the buzzphrase generator which we programmed back in Chap. 10. (You might want to review that version before reading this section.) The previous version did not keep track of its own carriage returns, causing us to force CRs into the definition and creating a very ragged right margin. The job of deciding how many whole words can fit on a line is a reasonable application for a computer and not a trivial one.

The problem is this: to draft a "brief" which consists of four paragraphs, each paragraph consisting of an appropriate introduction and sentence. Each sentence will consist of four randomly-chosen phrases linked together by fillers to create gramatically logical sentences and a period at the end.

The words and phrases have already been edited into blocks 234, 235, and 236 in the listing at the end of this section. Look at these blocks now, without looking at the two blocks that follow them (we're pretending we haven't written the application yet).

Block 234 contains the four introductions. They must be used in sequence. Block 235 contains four sets of fillers. The four sets must be used in sequence, but any of the three versions within a set may be chosen at random. Block 236 contains the three columns of buzzwords from our previous version, with some added words.

You might also look at the sample output that precedes the listing of the application, to get a better idea of the desired result.

"Top-down design" is a widely accepted approach to programming that can help to reduce development time. The idea is that you first study your application as a whole, then break the problem into smaller processes, then break these processes into still smaller units. Only when you know what all the units should do, and how they will connect together, do you begin to write code.

The FORTH language encourages top-down design. But in FORTH you can actually begin to write top-level definitions immediately. Already we can imagine that the "ultimate word" in our application might be called PAPER, and that it will probably be defined something like this:

```
: PAPER   4 0 DO I INTRO  SENTENCE LOOP ;
```

where INTRO uses the loop index as its argument to select the appropriate introduction. SENTENCE could be defined

```
: SENTENCE   4 0 DO I FILLER  PHRASE LOOP ;
```

where FILLER uses the loop index as its argument to select the appropriate set, then chooses at random one of the three versions within the set. The function of PHRASE will be the same as before.

Using FORTH's editor, we can enter these top-level definitions into a block. Of course we can't load the block until we have written our lower-level definitions.

In complicated applications, FORTH programmers often test the logic of their top-level definitions by using "stubs" for the lower-level words. A stub is a temporary definition. It might simply print a message to let us know its been executed. Or it may do nothing at all, except resolve the reference to its name in the high-level definition.

While the top-down approach helps to organize the programming process, it isn't always feasible to code in purely top-down fashion. Usually we have to find out how certain low-level mechanisms will work before we can design the higher-level definitions.

The best compromise is to keep a perspective on the problem as a whole while looking out for low-level problems whose solutions may affect the entire application.

In our example application, we can see that it will no longer be possible to force CRs at predictable points. Instead we've got to invent a mechanism whereby the computer will perform carriage returns automatically.

The only way to solve this problem is to count every character that is typed. Before each word is typed, the application must decide whether there is room to type it on the current line or do a carriage return first.

So let's define the variable LINECOUNT to keep the count and the constant RMARGIN with the value 78, to represent the maximum count per line. Each time we type a word we will add its count to LINECOUNT. Before typing each word we will execute this phrase:

```
(length of next word) LINECOUNT @ +  RMARGIN >  IF CR
```

that is, if the length of the next word added to the current length of the line exceeds our right margin, then we'll do a carriage return.

But we have another problem: how do we isolate words with a known count for each word? You got it, we use W[illegible].

Let's write out a "first draft" of this low-level part of our application. It will type a single word, making appropriate

calculations for carriage returns.

32 WORD	Finds one word delimited by a space.
COUNT DUP	Leaves the count and a copy of the count on the stack, with the address of the first character beneath.
LINECOUNT @ +	Computes how long the current line would be if the word were to be included on it.
RMARGIN >	Decides if it would exceed the margin.
IF CR 0 LINECOUNT !	If so, resets the carriage and the count.
ELSE SPACE THEN	Otherwise, leaves a space between the words.
DUP 1+ LINECOUNT +!	Increases the count by the length of the word to be typed, plus one for the space.
TYPE	Types the word using the count and the address left by COUNT.

Now the problem is getting [WORD] to look at the strings on disk. [WORD] gets its bearings from [BLK] and [>IN], so if we say,

```
234 BLK !   0 >IN !†
```

then [WORD] will begin scanning block 234, starting at the top (byte zero).

---

† For polyFORTH Users

The user variables [>IN] and [BLK] are adjacent to each other in the user table. This design allows you to fetch and store both together with [2@] and [2!]. For example,

```
234 0 >IN 2!
```

This causes another problem: by storing new values into the input stream pointers, we've destroyed the old values. If we now execute a definition that contains the above phrase, the interpreter will not come back to us when it's done; it will continue trying to interpret the rest of block 234. To solve this problem, our definition must save the pointer values somewhere before it changes them, then restore them just before it's done. Let's define a double-length variable called HOMEBASE, so we have a place to save the pointers. Then let's write a word whose job it will be to save the pointers in HOMEBASE. Finally, let's write a word which will restore the pointers.

```
VARIABLE HOMEBASE  2 ALLOT
: <WRITE   BLK @  >IN @  HOMEBASE 2! ;
: WRITE>   HOMEBASE 2@  >IN !  BLK ! ;
```

Now we have to modify our highest-level definition slightly, by editing in <WRITE at the beginning and WRITE> at the end:

```
: PAPER   <WRITE  4 0 DO I INTRO SENTENCE LOOP WRITE> ;
```

The next question is: how do we know when we've gotten to the end of the string?

Since we are typing word by word, we have to check whether >IN has advanced sixty-four places from its starting point every time we have found a new word. But the limit is not always sixty-four places; in the case of the buzzwords, the limit is twenty places.

For this reason, we should probably make the limit be an argument to a word. For example, the phrase

```
64 WORDS
```

should type out the contents of the 64-byte string, word by word, performing carriage returns where necessary.

How should we structure our definition of WORDS? Let's re-examine what it must do:

1. Determine whether there is still a word in the string to be typed.

2. If there is, type the word (with margin checking), then repeat. If there isn't, exit.

The two part nature of this structure suggests that we need a BEGIN...WHILE...REPEAT loop. Let's write our problem this way, if only to understand it better.

```
... BEGIN ANOTHER WHILE .WORD REPEAT ...
```

ANOTHER will do step 1; .WORD will do step 2.

How should ANOTHER determine whether there is still a word to be typed from the string? It must scan for the next word in the block, by using the phrase

```
32 WORD
```

then compare the new value of >IN against the limit for >IN, and finally return a "true" if the value is less than or equal to the limit. This flag will serve as the argument for WHILE.

How do we compute the limit for >IN? Before we can begin the above loop, we have to add the argument (sixty-four or whatever) to the beginning value of >IN and save this limit on the stack for ANOTHER to use each time through the loop. Thus our definition of WORDS might be

```
: WORDS   ( u -- )  >IN @ +  BEGIN ANOTHER WHILE
     .WORD REPEAT  2DROP ;
```

We need the 2DROP because, when we exit the loop, we will have the address of WORD's buffer and the limit for >IN on the stack, neither of which we need any longer.

Now we can define ANOTHER. We've already decided that the first thing it must do is find the next word, by using the phrase

```
32 WORD
```

At this point, there will be two values on the stack:

```
limit adr
```

We can perform the comparison with the phrase

```
OVER  >IN @  < NOT
```

By using OVER we save the limit on the stack for future loops. Remember that the phrase

```
< NOT
```

is the same as "greater than <u>or</u> equal to." Our definition of ANOTHER, then, might be

```
32 CONSTANT BL
: ANOTHER   ( limit -- limit adr)
     BL WORD  OVER  >IN @  < NOT ;
```

(The abbreviation BL is a common mnemonic[†] for "blank." We have used it here to improve program readability.)

How do we define .WORD? Actually, we've defined it already, a few pages back, with the exception that

```
32 WORD
```

should be omitted from the beginning of the definition, since it will have been performed in ANOTHER.

Now we have our word-typing mechanism. But let's see if we're overlooking anything. For example, consider that every time we start a new paragraph, we must remember to reset LINECOUNT to zero. Otherwise our .WORD will think that the current line is full when it isn't. We should ask ourselves this question: is there _ever_ a case in this application where we would want to perform a CR without resetting LINECOUNT? The answer is no, by the very nature of the application. For this reason we can define

```
: CR   CR 0 LINECOUNT ! ;
```

to create a version of CR that is appropriate for this application. We can use this CR in our definition of .WORD.

We should also consider our handling of spaces between words. By using the phrase

```
IF CR ELSE SPACE THEN
```

before typing each word, we guarantee that there will be a space between each pair of words on the same line but no space at the beginning of successive lines. And since we are typing a space _before_ each word rather than after, we can place a period immediately after a word, as we must at the end of a sentence.

But there's still a problem with this logic: at the beginning of a new paragraph, we will always get _one_ space before the first word. Our solution: to redefine SPACE so that it will be sensitive to whether or not we're at the beginning of a line, and will _not_ space if we _are_:

```
: SPACE   LINECOUNT @  IF SPACE THEN ;
```

If LINECOUNT is "0" then we know we are at the beginning of a line, because of the way we have redefined CR.

[†] For Beginners

As a general term, a "mnemonic" is a symbol or abbreviation chosen as an aid in remembering.

While we are redefining SPACE, it would be logical to include the phrase

```
1 LINECOUNT +!
```

in the redefinition. Again our reasoning is that we should never perform a space without incrementing the count. Now we can eliminate the word [1+] from the definition of .WORD, thereby eliminating a bug in the previous .WORD, namely that LINECOUNT was getting incremented even at the beginning of the line.

Let's assume that we have edited our definitions into a block. (In fact, we've done this already in block 237.) Notice that we had very little typing to do, compared with the amount of thinking we've done. FORTH source tends to be concise.†

Now we can define our in-between-level words--words like INTRO and PHRASE that we have already used in our highest-level words, but which we didn't define because we didn't have the low-level mechanism.

Let's start with INTRO. First we must set our input-stream pointers. The introductions are all in block 234, so the phrase

```
234 BLK !
```

takes care of them. Since each line is sixty-four bytes long, we can calculate the desired offset into the block by multiplying the loop index by sixty-four, then storing the offset into [>IN].

Now we're ready to use WORDS to type all the words in the next sixty-four bytes. The finished definition of INTRO looks like this:

```
: INTRO   ( u -- ) 64 * >IN !  234 BLK !  CR  64 WORDS ;
```

Our mechanism has given us a very easy way to select strings. Unfortunately we cannot test this definition by itself, because it does not reset the input-stream pointers to their original values when it's done. But we can get around this by writing ourselves a definition called TEST, as follows:

```
: TEST  CR  '  <WRITE EXECUTE WRITE> SPACE ;
```

Now we can say

---

†For Experts

On the other hand, FORTH is not as compressed as APL, which in our opinion is not nearly as readable as FORTH.

```
0 TEST INTRO

IN .::S PAPER WE WILL DEM(. . [RA ......AT ok
```

The "tick" in TEST will find the next word in the input stream, INTRO, which will then be executed "between" <WRITE and WRITE>. Notice that we put the argument to INTRO on the stack first.

The definition for FILLER will be a little more complicated. Since we are dealing with sets, not lines, and since the sets are four lines apart, we must multiply the loop index not by 64, but by (64 * 4). To pick one of the 3 versions <u>within</u> the set, we must choose a random number under three and <u>multiply</u> it by 64, then add this result to the beginning of the set. Recalling our discussion of compile-time arithmetic in Chap. 11., we can define

```
: FILLER   ( u -- ) [ 4 64 * ] LITERAL *
      3 CHOOSE  64 * +  >IN !  235 BLK !  64 WORDS ;
```

Again, we can test this definition by writing

```
3 TEST FILLER
TO FUNCTION AS ok
```

The remaining words in the application are similar to their previous counterparts, stated in terms of the new mechanism.

Here is a sample of the output, followed by our finished listing. (We've added block 239 as an afterthought so that we'd be able to print the same paper more than once.)

```
IN THIS PAPER WE WILL DEMONSTRATE THAT BY APPLYING AVAILABLE
RESOURCES TOWARDS FUNCTIONAL DIGITAL CAPABILITY COORDINATED WITH
COMPATIBLE ORGANIZATIONAL UTILITIES IT IS POSSIBLE FOR EVEN THE
MOST RESPONSIVE DIGITAL OUTFLOW TO AVOID TRANSIENT UNILATERAL
MOBILITY.

ON THE ONE HAND, STUDIES HAVE SHOWN THAT WITH STRUCTURED DEPLOYMENT
OF TOTAL FAIL-SAFE MOBILITY BALANCED BY SYSTEMATIZED UNILATERAL
THROUGH-PUT IT BECOMES NOT UNFEASABLE FOR ALL BUT THE LEAST RANDOM
ORGANIZATIONAL PROJECTIONS TO AVOID RESPONSIVE LOGISTICAL CONCEPTS.

ON THE OTHER HAND, HOWEVER, PRACTICAL EXPERIENCE INDICATES THAT
WITH STRUCTURED DEPLOYMENT OF QUALIFIED TRANSITIONAL MOBILITY
BALANCED BY REPRESENTATIVE LOGISTICAL THROUGH-PUT IT IS NECESSARY
FOR ALL REPRESENTATIVE UNILATERAL ENGINEERING TO FUNCTION AS
OPTIONAL DIGITAL SUPERSTRUCTURES.

IN SUMMARY, THEN, WE PROPOSE THAT WITH STRUCTURED DEPLOYMENT OF
RANDOM MANAGEMENT FLEXIBILITY BALANCED BY STAND-ALONE DIGITAL
CRITERIA IT IS NECESSARY FOR ALL QUALIFIED FAIL-SAFE OUTFLOW TO
AVOID PARTIAL UNDOCUMENTED ENGINEERING.
```

```
234 LIST

 0 IN THIS PAPER WE WILL DEMONSTRATE THAT
 1 ON THE ONE HAND, STUDIES HAVE SHOWN THAT
 2 ON THE OTHER HAND, HOWEVER, PRACTICAL EXPERIENCE INDICATES THAT
 3 IN SUMMARY, THEN, WE PROPOSE THAT
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15

235 LIST

 0 BY USING
 1 BY APPLYING AVAILABLE RESOURCES TOWARDS
 2 WITH STRUCTURED DEPLOYMENT OF
 3
 4 COORDINATED WITH
 5 TO OFFSET
 6 BALANCED BY
 7
 8 IT IS POSSIBLE FOR EVEN THE MOST
 9 IT BECOMES NOT-UNFEASABLE FOR ALL BUT THE LEAST
10 IT IS NECESSARY FOR ALL
11
12 TO FUNCTION AS
13 TO GENERATE A HIGH LEVEL OF
14 TO AVOID
15

236 LIST

 0 INTEGRATED           MANAGEMENT          CRITERIA
 1 TOTAL                ORGANIZATIONAL      FLEXIBILITY
 2 SYSTEMATIZED         MONITORED           CAPABILITY
 3 PARALLEL             RECIPROCAL          MOBILITY
 4 FUNCTIONAL           DIGITAL             PROGRAMMING
 5 RESPONSIVE           LOGISTICAL          CONCEPTS
 6 OPTIMAL              TRANSITIONAL        TIME PHASING
 7 SYNCHRONIZED         INCREMENTAL         PROJECTIONS
 8 COMPATIBLE           THIRD GENERATION    HARDWARE
 9 QUALIFIED            POLICY              THROUGH-PUT
10 PARTIAL              DECISION            ENGINEERING
11 STAND-ALONE          UNDOCUMENTED        OUTFLOW
12 RANDOM               CONTEXT SENSITIVE   SUPERSTRUCTURES
13 REPRESENTATIVE       FAIL-SAFE           INTERACTION
14 OPTIONAL             OMNIRANGE           CONGRUENCE
15 TRANSIENT            UNILATERAL          UTILITIES
```

```
237 LIST

 0 ( BUZZPHRASE GENERATOR II -- MARGIN SETTING)      EMPTY
 1 181 LOAD ( RANDOM NUMBERS)
 2 32 CONSTANT BL             78 CONSTANT RMARGIN
 3 VARIABLE LINECOUNT         VARIABLE HOMEBASE  2 ALLOT
 4 : <WRITE   BLK @  >IN @  HOMEBASE 2! ;
 5 : WRITE>   HOMEBASE 2@  >IN !  BLK ! ;
 6
 7 : CR   CR  0 LINECOUNT ! ;
 8 : SPACE   LINECOUNT @  IF SPACE  1 LINECOUNT +!  THEN ;
 9 : .WORD   ( adr)  COUNT DUP LINECOUNT @ + RMARGIN >
10           IF CR ELSE SPACE THEN
11           DUP  LINECOUNT +!  TYPE ;
12 : ANOTHER  ( lim -- lim adr)  BL WORD  OVER >IN @ < NOT ;
13 : WORDS   ( u)
14         >IN @ +  BEGIN ANOTHER WHILE .WORD REPEAT  2DROP ;
15 238 LOAD    239 LOAD

238 LIST

 0 ( BUZZPHRASE GENERATOR -- HIGH LEVEL WORDS)
 1
 2 : BUZZ   16 CHOOSE 64 * + >IN !  236 BLK !  20 WORDS ;
 3 : 1ADJ   0 BUZZ ;
 4 : 2ADJ   20 BUZZ ;
 5 : NOUN   40 BUZZ ;
 6 : PHRASE   1ADJ 2ADJ NOUN ;
 7 : FILLER   ( u)  [ 4 64 * ] LITERAL *
 8       3 CHOOSE 64 * + >IN !  235 BLK !    64 WORDS ;
 9 : SENTENCE   4 0 DO I FILLER PHRASE LOOP ." ." CR ;
10 : INTRO   ( u)   64 * >IN !  234 BLK !  CR 64 WORDS ;
11
12 : PAPER   <WRITE  CR CR  4 0 DO I INTRO SENTENCE LOOP WRITE> ;
13
14
15 : TEST   CR '  <WRITE EXECUTE WRITE> SPACE ;

239 LIST

 0 ( RETRIEVAL OF MORE SUCCESSFUL PAPERS)
 1
 2 VARIABLE SEED
 3
 4 : 4POSTERITY   RND @ SEED ! ;
 5 ( execute BEFORE producing a paper)
 6
 7 : REDO   SEED @ RND ! ;
 8 ( execute AFTER a paper, to reprint it.
 9   Usage:  REDO PAPER    )
10
11
12
13
14
15
```

Copyright FORTH, Inc.   3/06/81   11:44   Starting FORTH

File Away!

Our second example consists of a simple filing system.† It is a powerful and useful application, and a good one to learn FORTH style from. We have divided this section into four parts:

1. A "How to" for the end user. This will give you an idea of what the application can do.
2. Notes on the way the application is structured and the way certain definitions work.
3. A glossary of all the definitions in the application.
4. A listing of the application, including the blocks that contain the files themselves.

How to Use the Simple File System

This computer filing system lets you store and retrieve information quickly and easily. At the moment, it is set up to handle people's names, occupations, and phone numbers.‡ Not only does it allow you to enter, change, and remove records, it also allows you to search the file for any piece of information. For example, if you have a phone number, you can find the person's name; or, given a name, you can find the person's job, etc.

For each person there is a "record" which contains four "fields." The names which specify each of these four fields are

SURNAME GIVEN JOB PHONE

("Given," of course, refers to the person's given name, or first name.)

---

† For Serious File-Users

FORTH, Inc. offers a very powerful File Management Option.

‡ For Programmers

You can easily change these categories or extend the number of fields the system will handle.

File Retrieval

You can search the file for the contents of any field by using the word FIND, followed by the field-name and the contents, as in

FIND JOB NEWSCASTER RETURN DAN RATHER ok

If any "job" field contains the string "NEWSCASTER," then the system prints the person's full name. If no such file exists, it prints "NOT IN FILE."

Once you have found a field, the record in which it was found becomes "current." You can get the contents of any field in the current record by using the word GET. For instance, having entered the line above, you can now enter

GET PHONE RETURN [illegible]-9876 ok

The FIND command will only find the first instance of the field that you are looking for. To find out if there is another instance of the field that you last found, use the command ANOTHER. For example, to find another person whose "job" is "NEWSCASTER," enter

ANOTHER RETURN JESSICA SAVITCH ok

and

ANOTHER RETURN FRANK REYNOLDS ok

When there are no more people whose job is "NEWSCASTER" in the file, the ANOTHER command will print "NO OTHER."

To list all names whose field contains the string that was last found, use the command ALL:

ALL RETURN
[illegible]HER
[illegible] SAVIT[illegible]
FR[illegible] [illegible]NOL[illegible]
ok

Since the surname and given name are stored separately, you can use FIND to search the file on the basis of either one. But if you know the person's full name, you can often save time by locating both fields at once, by using the word FULLNAME. FULLNAME expects the full name to be entered with the last name first and the two names separated by a comma, as in

FULLNAME WONDER,STEVIE RETURN STEVIE WONDER ok

(There must not be a space after the comma, because the comma marks the end of the first field and the beginning of the second field.) Like FIND and ANOTHER, FULLNAME repeats the name to indicate that it has been found.

You can actually find any pair of fields by using the word PAIR. You must specify both the field names and their contents, separated by a comma. For example, to find a newscaster whose given name is Dan, enter

```
PAIR JOB NEWSCASTER,GIVEN DAN RETURN DAN RATHER ok
```

File Maintenance

To enter a new record, use the command ENTER, followed by the surname, given name, job, and phone, each separated by a comma only. For example,

```
ENTER NUREYEV,RUDOLF,BALLET DANCER,555-1234 RETURN ok
```

To change the contents of a single field within the current record, use the command CHANGE followed by the name of the field, then the new string. For example,

```
CHANGE JOB CHOREOGRAPHER RETURN ok
```

To completely remove the current record, use the command REMOVE:

```
REMOVE RETURN ok
```

After adding, changing, or removing records, and before turning off the computer or changing disks, be sure to use the word

```
FLUSH ok
```

Comments

This section is meant as a guide, for the novice FORTH programmer, to the glossary and listing which follow. We'll describe the structure of this application and cover some of the more complicated definitions. As you read this section, study the glossary and listing on your own, and try to understand as much as you can.

Turn to the listing now and look at block 242. This block contains the definitions for all nine end-user commands we've just discussed. Notice how simple these definitions are, compared to their power!

This is a characteristic of a well-designed FORTH application. Notice that the word -FIND, the elemental file-search word, is factored in such a way that it can be used in the definitions of FIND, ANOTHER, and ALL, as well as in the internal word, (PAIR), which is used by PAIR and by FULLNAME.

We'll examine these definitions shortly, but first let's look at the overall structure of this application.

One of the basic characteristics of this application is that each of the four fields has a name which we can enter in order to specify the particular field. For example, the phrase

```
SURNAME PUT
```

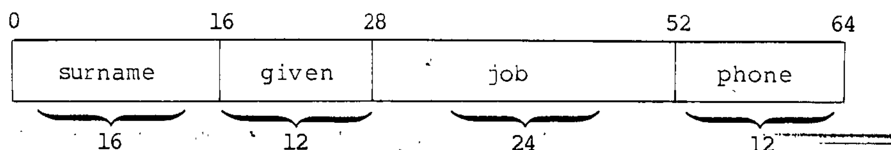
will put the character string that follows in the input stream into the "surname" field of the current record. The phrase

```
SURNAME .FIELD
```

will print the contents of the "surname" field of the current record, etc.

There are two pieces of information that are needed to identify each field: the field's starting address relative to the beginning of a record and the length of the field.

In this application, a record is laid out like this:



For instance, the "job" field starts twenty-eight bytes in from the beginning of every record and continues for twenty-four bytes.

We chose to make a record exactly sixty-four bytes long so that the fields will line up in columns when we LIST the file. This was for our convenience in programming, but this system could be

modified to hold records of any length and any number of fields.†

We've taken the two pieces of information for each field and put them into a double-length table associated with each field name. Our definition of JOB, therefore, is

```
CREATE JOB  28 , 24 ,
```

3	J
O	B
link	
code pointer	
28	
24	

Thus when we enter the name of a field, we are putting on the stack the address of the table that describes the "job" field. We can fetch either or both pieces of information relative to this address.

Let's call each of these entries a "field specifying table," or a "spec table" for short.

---

† For Those Who Want to Modify This File System

To change the parameters of the fields, just make sure that the beginning byte ("tab") for each field is consistent with the lengths of the fields that precede it. For example, if the first field is thirty bytes long, as in

```
CREATE 1FIELD  0 , 30 ,
```

then make the tab for the second field thirty, as in

```
CREATE 2FIELD  30 , 12 ,
```

etc. Finally, set the value of R-LENGTH in line 4 to the length of the entire record (the last field's tab plus its length). Using R-LENGTH, the system automatically computes the number of records that can fit into a single block (1024 R-LENGTH /) and defines the constant REC/BLK accordingly.

You may also change the location of the new file (e.g., to create several different files) by changing the value of the constant FILE in line 5. You may also change the maximum number of blocks that your file can contain by replacing the "2" in the same line. This value will be converted into a maximum number of records, by being multiplied by REC/BLK, and kept as the constant MAXRECS.

Part of the design for this application is derived from the requirements of FIND, ANOTHER, and ALL; that is, FIND not only has to find a given string within a given type of field, but also needs to "remember" the string and the type of field so that ANOTHER and ALL can search for the same thing.

We can specify the kind of field with just one value, the address of the spec table for that type of field. This means that we can "remember" the type of field by storing this address into KEEP.

KIND was created for this purpose, to indicate the "kind" of field.

To remember the string, we have defined a buffer called WHAT to which the string can be moved. (WHAT is defined relative to the pad, where memory can be reused, so as not to waste dictionary space.)

The word KEEP serves the dual purpose of storing the given field type into KIND and the given character string into WHAT. If you look at the definition of the end-user word FIND, you will see that the first thing it does is KEEP the information on what is being searched for. Then FIND executes the internal word -FIND, which uses the information in KIND and WHAT to find a matching string.

ANOTHER and ALL also use -FIND, but they don't use KEEP. Instead they look for fields that match the one most recently "kept" by FIND.

So that we can GET any piece of information from the record which we have just "found," we need a pointer to the "current" record. This need is met with the variable #RECORD. The operations of the words TOP and DOWN in block 240 should be fairly obvious to you.

The word RECORD uses #RECORD to compute the absolute address (the computer-memory address, somewhere in a disk buffer) of the beginning of the current record. Since RECORD executes BLOCK, it also guarantees that the record really is in a buffer.

Notice that RECORD allows the file to continue over a range of blocks. /MOD divides the value of #RECORD by the number of records per block (sixteen in this case, since each record is sixty-four bytes long). The quotient indicates which block the record will be in, relative to the first block; the remainder indicates how far into that block this record will be.

While a spec table contains the relative address of the field and its length, we usually need to know the field's absolute address and length for words such as TYPE, MOVE, and -TEXT. Look at the definition of the word FIELD to see how it converts the address of a spec table into an absolute address and length.

Then examine how FIELD is applied in the definition of .FIELD.

The word PUT also employs FIELD. Its phrase

PAD SWAP FIELD

leaves on the stack the arguments

adr-of-PAD absolute-adr-of-field count

for MOVE to move the string from the pad into the appropriate field of the current record.

There are two things worth noting about the definition of FREE in block 241. The first is the method used to determine whether a record is empty. We've made the assumption that if the first byte of a record is empty, then the whole record is empty, because of the way ENTER works. If the first byte contains a character whose ASCII value is less than thirty-three (thirty-two is blank), then it is not a printing character and the line is empty. (Sometimes an empty block will contain all nulls, other times all blanks; either way, such records will test as "empty.") As soon as an empty record is found, LEAVE ends the loop. #RECORD will contain the number of the free record.

Another thing worth noting about FREE is that it aborts if the file is full, that is, if it runs through all the records without finding one empty. We can use a DO loop to run through all the records, but how can we tell that the loop has run out before it has found an empty record?

The best way is to leave a "1" on the stack, to serve as a flag, before beginning the loop. If an empty record is found, we can change the flag to zero (with the word NOT) before we leave the loop. When we come out of the loop, we'll have a "1" if we never found an empty record, a "0" if we did. This flag will be the argument for ABORT".

We use a similar technique in the definition of -FIND. -FIND must return a flag to the word that executed it: FIND, ANOTHER, ALL, or (PAIR). The flag indicates whether a match was found before the end of the file was reached. Each of these outer words needs to make a different decision based on the state of this flag. This flag is a "1" if a match is not found (hence the name -FIND). The decision to use negative logic was based on the way -FIND is used.

Because the flag needs to be a "1" if a match is not found, the easiest way to design this word is to start with a "1" on the stack and change it to a "0" only if a match is found. But notice: while the loop is running, there are two values on the stack: the flag we just mentioned and the spec table address for the type of field to be searched. Since we need the address

every time through the loop and the flag only once, if at all, we have decided to keep the address on top of the stack and the flag underneath. For this reason, we use the phrase

```
SWAP NOT SWAP
```

By the way, we could have avoided the problem of carrying both values on the stack by putting the phrase

```
KIND @ FIELD
```

inside the loop, instead of

```
KIND @
```

at the beginning and

```
DUP FIELD
```

inside. But we didn't, because we always try to keep the number of instructions inside a loop to a minimum. Naturally, it is the loops that take the most time running.

Now that you understand the basic design of this application, you should have no trouble understanding the rest of the listing, using the glossary as a guide.†

---

† For polyFORTH Users

This type of glossary is generated by an application called DOCUMENTOR, which is included in the File Management Option.

```
FORTH, Inc.                                                  Page 1    3/06/81
                SIMPLE FILES GLOSSARY
 WORD           VOCABULARY   BLOCK   STACK EFFECTS


 #RECORD        FORTH           240     ( -- adr)
     A variable that points to the current record.

 (PAIR)         FORTH           241     ( adr)
     Starting from the top, attempts to find a match on the contents
     of WHAT, using KIND to indicate the type of field.  If a match
     is made, then attempts to match a second field, whose type is
     indicated by adr, with the contents of PAD.  If both match,
     prints the name; otherwise repeats until a match is made or
     until the end of the file is reached, in which case prints
     an error message.

 -FIND          FORTH           241     ( -- f)
     Beginning with #RECORD and proceeding down, compares the contents
     of the field indicated by KIND against the contents of WHAT.

 .FIELD         FORTH           240     ( adr)
     From the current record, types the contents of the field that is
     associated with the field-specifying table at adr.

 .NAME          FORTH           240
     From the current record, types the name, first name first.

 ALL            FORTH           242
     Beginning at the top of the file, uses KIND to determine type of
     field and finds all matches on WHAT.  Types the full name(s).

 ANOTHER        FORTH           242
     Beginning with the next record after the current one, and using
     KIND to determine type of field, attempts to find a match on WHAT.
     If successful, types the name; otherwise an error message.

 CHANGE         FORTH           242
     Changes the contents of the given field in the current record.
     usage:  CHANGE  field-name  new-contents

 DOWN           FORTH           240
     Moves the record pointer down one record.

 ENTER          FORTH           242
     Finds the first free record, then moves four strings separated
     by commas into the surname, given, job, and phone fields of
     that record.

 FIELD          FORTH           240     ( adr -- adr length)
     Given the address of a field-specifying table, insures that
     the associated field in the current record is in a disk buffer
     and returns the address of the field in the buffer along with
     its length.

 FILES          FORTH           240     ( -- u)
     The number of the block where the files begin.
```

```
FORTH, Inc.                                          Page 2   3/06/81
              SIMPLE FILES GLOSSARY
WORD          VOCABULARY   BLOCK   STACK EFFECTS

FIND          FORTH        242
    Finds the record in which there is a match between the contents
    of the given field and the given string.
       Usage:  FIND  field-name  string

FREE          FORTH        241
    Starting at the top of the file, finds the first record that is
    free, that is, whose first byte contains a blank or zero.
    Aborts if the file is full.

FULLNAME      FORTH        242
    Finds the record in which there is a match on both the first and
    last names given.   Usage:  FULLNAME lastname,firstname

GET           FORTH        242
    Prints the contents of the given type of field from the current
    record.

GIVEN         FORTH        240     ( -- adr)
    Returns the address of the field-specifying table for the
    "given" (first name) field.

JOB           FORTH        240     ( -- adr)
    Returns the address of the field-specifying table for the
    "job" field.

KEEP          FORTH        241     ( adr)
    Moves a character string, delimited either by a comma or by a
    carriage return, from the input stream into WHAT, and saves the
    address of the given field-specifying table in KIND, for future
    use by -FIND.

KIND          FORTH        240     ( -- adr)
    A variable that contains the address of the field-specifying
    table for the type of field that was last searched for by FIND.

MAXRECS       FORTH        240     ( -- u)
    The maximum number of records to be allowed in the system.

MISSING       FORTH        241
    Prints the message "NOT IN FILE."

PAIR          FORTH        242
    Finds the record in which there is a match between both the
    contents of the first given field and the first given string, and
    and also the contents of the second given field and the second
    given string.  Comma is the delimiter.
       Usage:    PAIR  field1 string1,field2 string2

PHONE         FORTH        240     ( -- adr)
    Returns the address of the field-specifying table for the
    "phone" field.
```

FORTH, Inc. Page 3 3/06/81

SIMPLE FILES GLOSSARY

WORD	VOCABULARY	BLOCK	STACK EFFECTS

PUT FORTH 241 (adr)
Moves a character string, delimited either by a comma or by a carriage return, from the input stream into the field whose field-specifying-table address is given on the stack.

R-LENGTH FORTH 240 (-- u)
The length in bytes of a single record.

READ FORTH 241
Moves a character string, delimited either by a comma or by a carriage return, from the input stream into PAD.

REC/BLK FORTH 240 (-- u)
The number of records that will fit in a single block, given MAXRECS.

RECORD FORTH 240 (-- adr)
Insures that the current record is in a disk buffer, and returns the address of the first byte of that record.

REMOVE FORTH 242
Erases the current record.

SURNAME FORTH 240 (-- adr)
Returns the address of the field-specifying table for the "surname" (last name) field.

TOP FORTH 240
Resets the record pointer to the top of the file.

WHAT FORTH 240 (-- adr)
Returns the address of a buffer that contains the string that is being searched for, or was last searched for, by FIND.

```
240 LIST

 0 ( SIMPLE FILES)                                 EMPTY
 1                ( tab length)                          ( tab length)
 2 CREATE SURNAME   0 ,  16 ,         CREATE GIVEN  16 ,  12 ,
 3 CREATE JOB      28 ,  24 ,         CREATE PHONE  52 ,  12 ,
 4 64 CONSTANT R-LENGTH          1024 R-LENGTH / CONSTANT REC/BLK
 5 243 CONSTANT FILES                 2 REC/BLK * CONSTANT MAXRECS
 6 VARIABLE #RECORD                     VARIABLE KIND
 7 : WHAT   ( -- adr)  PAD 80 + ;
 8 : RECORD   ( -- first adr of current record)
 9     #RECORD @ REC/BLK /MOD  FILES + BLOCK  SWAP R-LENGTH * + ;
10 : FIELD   ( field --adr length)  2@ RECORD +  SWAP ;
11 : TOP   0 #RECORD ! ;
12 : DOWN   1 #RECORD +! ;
13 : .FIELD   ( field)  FIELD  -TRAILING TYPE  SPACE ;
14 : .NAME   GIVEN .FIELD ' SURNAME .FIELD ;
15 241 LOAD  242 LOAD


241 LIST

 0 ( SIMPLE FILES, CONT'D)
 1 : READ   44 TEXT ;
 2 : PUT   ( field)  READ  PAD SWAP FIELD MOVE  UPDATE ;
 3 : KEEP   ( field) DUP KIND !
 4                    2+ @  READ  PAD WHAT ROT MOVE ;
 5 : FREE   1  MAXRECS 0 DO  I #RECORD !  RECORD C@
 6     ( ASCII) 33 <  IF NOT LEAVE THEN  LOOP ABORT" FILE FULL " ;
 7
 8 : -FIND   ( -- f)   1  KIND @  MAXRECS  #RECORD @ DO
 9          I #RECORD !  DUP FIELD WHAT -TEXT NOT IF
10             SWAP NOT SWAP  LEAVE THEN  LOOP  DROP ;
11 : MISSING   ." NOT IN FILE " ;
12 : (PAIR)   ( field)    MAXRECS 0 DO  I #RECORD !
13     -FIND IF  MISSING LEAVE ELSE  DUP FIELD PAD -TEXT NOT
14                    IF .NAME LEAVE THEN   THEN   LOOP DROP ;
15


242 LIST

 0 ( SIMPLE FILES -- END USER WORDS)
 1
 2 : ENTER   FREE   SURNAME PUT    GIVEN PUT
 3                  JOB PUT        PHONE PUT ;
 4 : REMOVE   RECORD R-LENGTH 32 FILL  UPDATE ;
 5 : CHANGE   '  PUT ;
 6
 7 : FIND   '  KEEP  TOP -FIND  IF MISSING ELSE .NAME THEN ;
 8 : GET   '  .FIELD ;
 9
10 : ANOTHER   DOWN -FIND  IF ." NO OTHER "  ELSE .NAME THEN ;
11 : ALL   TOP BEGIN  CR -FIND NOT  WHILE  .NAME DOWN  REPEAT ;
12
13 : PAIR   '  KEEP   '  READ (PAIR) ;
14 : FULLNAME   SURNAME KEEP  GIVEN READ (PAIR) ;
15
```

Copyright FORTH, Inc.   3/06/81 11:44   Starting FORTH

```
243 LIST

  0 FILLMORE      MILLARD     PRESIDENT                 NO PHONE
  1 LINCOLN       ABRAHAM     PRESIDENT                 NO PHONE
  2 BRONTE        EMILY       WRITER                    NO PHONE
  3 RATHER        DAN         NEWSCASTER                555-9876
  4 FITZGERALD    ELLA        SINGER                    555-6789
  5 SAVITCH       JESSICA     NEWSCASTER                555-9653
  6 MC CARTNEY    PAUL        SONGWRITER                555-1212
  7 WASHINGTON    GEORGE      PRESIDENT                 NO PHONE
  8 REYNOLDS      FRANK       NEWSCASTER                555-8765
  9 SILLS         BEVERLY     OPERA STAR                555-9876
 10 FORD          HENRY       CAPITALIST                NO PHONE
 11 DEWHURST      COLEEN      ACTRESS                   555-9876
 12 WONDER        STEVIE      SONGWRITER                555-0097
 13 FULLER        BUCKMINSTER WORLD ARCHITECT           555-7604
 14 RAWLES        JOHN        PHILOSOPHER               555-9721
 15 TRUDEAU       GARRY       CARTOONIST                555-9832

244 LIST

  0 VAN BUREN     ABIGAIL     COLUMNIST                 555-8743
  1 ABZUG         BELLA       POLITICIAN                555-4443
  2 THOMPSON      HUNTER S.   GONZO JOURNALIST          555-9854
  3 SINATRA       FRANK       SINGER                    555-9412
  4 JABBAR        KAREEM ABDULBASKETBALL PLAYER         555-4439
  5 MC GEE        TRAVIS      FICTITIOUS DETECTIVE      555-8887
  6 DIDION        JOAN        WRITER                    555-0009
  7 FRAZETTA      FRANK       ARTIST                    555-9991
  8 HENSON        JIM         PUPPETEER                 555-0001
  9
 10
 11
 12
 13
 14
 15

245 LIST

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

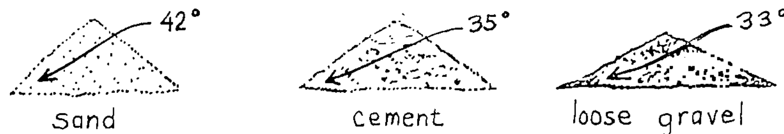
Copyright FORTH, Inc.   3/06/81   11:44   Starting FORTH

No Weighting

Our final example is a math problem which many people would assume could only be solved by using floating point. It will illustrate how to handle a fairly complicated equation with fixed-point arithmetic and demonstrate that for all the advantages of using fixed-point, range and precision need not suffer.

In this example we will compute the weight of a cone-shaped pile of material, knowing the height of the pile, the angle of the slope of the pile, and the density of the material.

To make the example more "concrete," let's weigh several huge piles of sand, gravel, and cement. The slope of each pile, called the "angle of repose," depends on the type of material. For example, sand piles itself more steeply than gravel.



(In reality these values vary widely, depending on many factors; we have chosen approximate angles and densities for purposes of illustration.)

Here is the formula for computing the weight of a conical pile h feet tall with an angle of repose of $\underline{\theta}$ degrees, where D is the density of the material in pounds per cubic foot:†

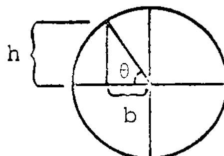
---

†For Skeptics

The volume of a cone, V, is given by

$$V = \frac{1}{3}\pi b^2 h$$

where b is the radius of the base and h is the height. We can compute the base by knowing the angle or, more specifically, the tangent of the angle. The tangent of an angle is simply the ratio of the segment marked h to the segment marked b in this drawing:



(continued...)

$$W = \frac{\pi h^3 D}{3 \tan^2(\theta)}$$

This will be the formula which we must express in FORTH.

Let's design our application so that we can enter the name of a material first, such as

DRY-SAND

then enter the height of a pile and get the result for dry sand.

Let's assume that for any one type of material the density and angle of repose never vary. We can store both of these values for each type of material into a table. Since we ultimately need each angle's tangent, rather than the number of degrees, we will store the tangent. For instance, the angle of repose for a pile of cement is 35°, for which the tangent is .700. We will store this as the integer 700.

CEMENT
131
700

Bear in mind that our goal is not just to get an answer; we are programming a computer or device to get the answer for us in the fastest, most efficient, and most accurate way possible. As we indicated in Chap. 5, to write equations using fixed-point arithmetic requires an extra amount of thought. But the effort pays off in two ways:

For Skeptics (continued)

If we call this angle "θ" (theta), then

$$\tan \theta = \frac{h}{b}$$

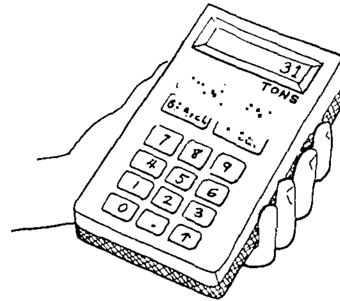
Thus we can compute the radius of the base with

$$b = \frac{h}{\tan \theta}$$

When we substitute this into the expression for V, and then multiply the result by the density D in pounds per cubic foot, we get the formula shown above.

1. vastly improved run-time speed, which can be very important when there are millions of steps involved in a single calculation, or when we must perform thousands of calculations every minute. Also,

2. program size, which would be critical if, for instance, we wanted to put this application in a hand-held device specifically designed as a pile-measuring calculator. FORTH is often used in this type of instrument.



Let's approach our problem by first considering scale. The height of our piles ranges from 5 to 50 feet. By working out our equation for a pile of cement 50 feet high, we find that the weight will be nearly 35,000,000 pounds.

But because our piles will not be shaped as perfect cones and because our values are averages, we cannot expect better than four or five decimal places of accuracy.† If we scale our result to tons, we get about 17,500. This value will comfortably fit within the range of a single-length number. For this reason, let's write this application entirely with single-length arithmetic operators.

Applications which require greater accuracy can be written using double-length arithmetic; to illustrate we've even written a second version of <u>this</u> application using 32-bit math, as you'll see later on. But we intend to show the accuracy that FORTH can achieve even with 16-bit math.

By running another test with a pile 40 feet high, we find that a difference of one-tenth of a foot in height can make a difference of 25 tons in weight. So we decide to scale our input to feet and inches rather than merely to whole feet.

† For Math Experts:

In fact, since our height will be expressed in three digits, we can't expect greater than three-digit precision. But for purposes of our example, we'll keep better than four-digit precision.

We'd like the user to be able to enter

```
15 FOOT  2 INCH  PILE
```

--where the words FOOT and INCH will convert the feet and inches into tenths of an inch, and PILE will do the calculation. Here's how we might define FOOT and INCH:

```
: FOOT   10 * ;
: INCH   100 12 */ 5 +  10 / + ;
```

The use of INCH is optional.

(By the way, we could as easily have designed input to be in tenths of an inch with a decimal point, like this:

```
15.2
```

In this case, NUMBER would convert the input as a double-length value. Since we are only doing single-length arithmetic, PILE could simply begin with [DROP], to eliminate the high-order byte.)

In writing the definition of PILE, we must try to maintain the maximum number of places of precision without overflowing 15 bits. According to the formula, the first thing we must do is cube the argument. But let's remember that we will have an argument which may be as high as 50 feet, which will be 500 as a scaled integer. Even to square 500 produces 250,000, which exceeds the capacity of single-length arithmetic.

We might reason that, sooner or later in this calculation, we're going to have to divide by 2000 to yield an answer in tons. Thus the phrase

```
DUP DUP 2000 */
```

will square the argument and convert it to tons at the same time, taking advantage of [*/]'s double-length intermediate result. Using 500 as our test argument, the above phrase will yield 125.

But our pile may be as small as 5 feet, which when squared is only 25. To divide by 2000 would produce a zero in integer arithmetic, which suggests that we are scaling down too much.

To retain maximum accuracy, we should scale down no more than necessary. 250,000 can be safely accommodated by dividing by 10. Thus we will begin our definition of PILE with the phrase

```
DUP DUP 10 */
```

The integer result at this stage will be scaled to one place to the right of the decimal point (25000 for 2500.0).

Now we must cube the argument. Once again, straight multiplication will produce a double-length result, so we must use */ to scale down. We find that by using 1000 as our divisor, we can stay just within single-length range. Our result at this stage will be scaled to one place to the left of the decimal point (12500 for 125000.) and still accurate to 5 digits.

According to our formula, we must multiply our argument by pi. We know that we can do this in FORTH with the phrase

```
355 113 */
```

We must also divide our argument by 3. We can do both at once with the phrase

```
355 339 */
```

which causes no problems with scaling.

Next we must divide our argument by the tangent squared, which we can do by dividing the argument by the tangent twice. Because our tangent is scaled to 3 decimal places, to divide by the tangent we multiply by 1000 and divide by the table value. Thus we will use the phrase

```
1000 THETA @ */
```

Since we must perform this twice, let's make it a definition, called /TAN (for divide-by-the-tangent) and use the word /TAN twice in our definition of PILE. Our result at this point will still be scaled to one place to the left of the decimal (26711 for 267110, using our maximum test values).

All that remains is to multiply by the density of the material, of which the highest is 131 pounds per cubic foot. To avoid overflowing, let's try scaling down by two decimal places with the phrase

```
DENSITY @ 100 */
```

But by testing, we find that the result at this point for a 50-foot pile of cement will be 34,991, which just exceeds the 15-bit limit. Now is a good time to take the 2000 into account. Instead of

```
DENSITY @ 100 */
```

we can say

```
DENSITY @ 200 */
```

and our answer will now be scaled to whole tons.

You will find this version in the listing of block 246 that

follows. As we mentioned, we have also written this application using double-length arithmetic, in block 248. In this version you enter the height as a double-length number scaled to tenths of a foot, followed by the word FEET, as in 50.0 feet.

By using double-length integer arithmetic, we are able to compute the weight of the pile to the nearest whole pound. The range of double-length integer arithmetic compares with that of most floating-point arithmetic. Below is a comparison of the results obtained using a 10-decimal-digit calculator, single-length FORTH, and double-length FORTH. The test assumes a 50-foot pile of cement, using the table values.

	in pounds	in tons
calculator	34,995,634	17,497.817
FORTH 16-bit	---	17,495
FORTH 32-bit	34,995,634	17,497.817

Here's a sample of our application's output:

```
246 LOAD ok
CEMENT ok
10 FOOT PILE = 138 T[illegible] OF CEM[illegible] ok
10 FOOT 3 INCH PILE = [illegible]1 TONS [illegible] CEMENT ok
DRY-SAND ok
10 FOOT PILE = 81 TONS OF DRY SAND ok
248 LOAD CEMENT ok
10.0 FEET = 279939 POUNDS OF CEMENT OR 139.969 T[illegible] ok
```

A note on "

The defining word MATERIAL takes three arguments for each material, one of which is the address of a string. .SUBSTANCE uses this address to type the name of the material.

To put the string in the dictionary and to give an address to MATERIAL, we have defined a word called ". As you can see from its definition, " compiles the string (delimited by a second quotation mark, ASCII 34) into the dictionary, with the count in the first byte, and leaves its address on the stack for MATERIAL. To compile the count and string into the dicti[illegible]ary, we simply have to execute WORD, since WORD's buffer is [illegible]. We get the string's address as a fillip, since WORD also leaves [illegible]

All that remains is to ALLOT the appropriate number of bytes. This number is obtained by fetching the count from the first byte of the string and adding one for the count's byte.

```
246 LIST

     0 ( WEIGHT OF CONICAL PILES -- SINGLE-LENGTH)      EMPTY
     1 VARIABLE DENSITY   VARIABLE THETA   VARIABLE STRING
     2 34 CONSTANT QUOTE
     3 : "   QUOTE WORD  DUP C@ 1+ ALLOT ;
     4 : .SUBSTANCE   STRING @  COUNT  TYPE SPACE ;
     5
     6 : MATERIAL   ( STRING DENSITY THETA) CREATE  , , ,
     7      DOES> DUP @ THETA !  2+ DUP @ DENSITY !  2+ @ STRING ! ;
     8
     9 : FOOT   10 * ;
    10 : INCH   100 12 */  5 + 10 /  + ;
    11
    12 : /TAN   1000 THETA @  */ ;
    13 : PILE   DUP DUP 10 */  1000 */  355 339 */ /TAN /TAN
    14         DENSITY @  200 */  ." = "  .  ." TONS OF " .SUBSTANCE ;
    15 247 LOAD

247 LIST

     0 ( TABLE OF MATERIALS)
     1 ( STRING-ADDRESS   DENSITY  THETA)
     2 " CEMENT"             131      700   MATERIAL CEMENT
     3 " LOOSE GRAVEL"        93      649   MATERIAL LOOSE-GRAVEL
     4 " PACKED GRAVEL"      100      700   MATERIAL PACKED-GRAVEL
     5 " DRY SAND"            90      754   MATERIAL DRY-SAND
     6 " WET SAND"           118      900   MATERIAL WET-SAND
     7 " CLAY"               120      727   MATERIAL CLAY
     8
     9
    10
    11
    12
    13
    14 CEMENT
    15

248 LIST

     0 ( WEIGHT OF CONICAL PILES -- DOUBLE-LENGTH)      EMPTY
     1 VARIABLE DENSITY   VARIABLE THETA   VARIABLE STRING
     2 34 CONSTANT QUOTE
     3 : "   QUOTE WORD  DUP C@ 1+ ALLOT ;
     4 : .SUBSTANCE   STRING @  COUNT  TYPE SPACE ;
     5 : U.3   <#  # # # 46 HOLD #S  #>  TYPE SPACE ;
     6 : MATERIAL   ( STRING DENSITY THETA) CREATE  , , ,
     7      DOES> DUP @ THETA !  2+ DUP @ DENSITY !  2+ @ STRING ! ;
     8
     9 : CUBE   ( d -- d)  2DUP OVER 10 M*/  DROP  10 M*/ ;
    10 : /TAN   ( d -- d)  1000 THETA @  M*/ ;
    11 : FEET   ( d -- d)   CUBE  355 339 M*/  DENSITY @ 1 M*/
    12             /TAN  /TAN 5 M+  1 10 M*/
    13           2DUP  ." = " D. ." POUNDS OF "  .SUBSTANCE
    14         1 2 M*/   ." OR " U.3  ." TONS " ;
    15 247 LOAD
```

Review of Terms

Stub	in FORTH, a temporary definition created solely to allow testing of a higher-level definition.
Top-down programming	a programming methodology by which a large application is divided into smaller units, which may be further subdivided as necessary. The design process starts with the overview, or "top," and proceeds down to the lowest level of detail. Coding of the low-level units begins only after the entire structure of the application has been designed.