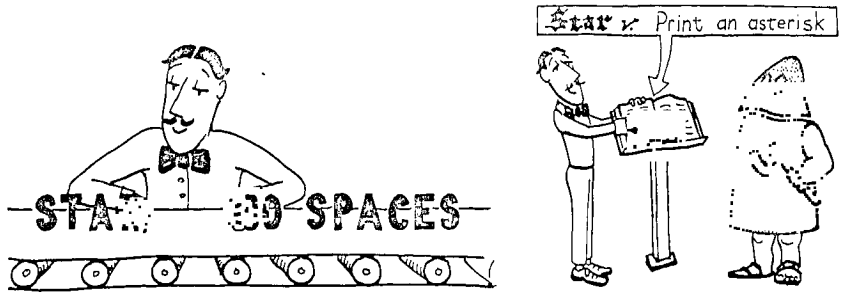
9 UNDER THE HOOD

Let's stop for a chapter to lift FORTH's hood and see what goes on inside.

Some of the information contained herein we've given earlier, but, at the risk of redundancy, we're now going to view the FORTH "machine" as a whole, to see how it all fits together.

Inside [illegible]

Back in the first chapter we learned that the text interpreter, whose name is [INTERPRET], picks words out of the input stream and tries to find their definitions in the dictionary. If it finds a word, [INTERPRET] has it executed.



We can perform these separate operations ourselves by using words that perform the component functions of [INTERPRET]. For instance, the word ['] (an apostrophe, but pronounced tick) finds a definition in the dictionary and returns its address. If we have defined GREET as we did in Chap. 1, we can now say

```
' GREET U. 25520 ok
```

and discover the address of GREET (whatever it happens to be).

We may also directly use EXECUTE. EXECUTE will execute a definition, given its address on the stack. Thus we can say

```
' GREET EXECUTE HELLO I SPEAK FORTH ok
```

and accomplish the same thing as if we had merely said GREET, only in a more roundabout way.

If tick cannot find a word in the dictionary, it executes ABORT" and prints a question mark.

FORTH's text interpreter uses a word related to tick that returns a zero flag if the word is found. The name and usage of the word varies,†‡ but the conditional structure of the [IN . :] phrase always looks like this:

```
(find the word)    IF      (convert to a number)
                   ELSE    (execute the word)
                   THEN
```

that is, if the string is not a defined word in the dictionary, INTERP .] tries to convert it as a number. If it is a defined word, I. :RPRET executes it.

The word ['] has several uses. For instance, you can use the phrase

```
' GREET .
```

to find out whether GREET has been defined, without actually having to execute it (it will either print the address or respond "?"). In systems that only save the first three characters of a name, you can also use the above phrase to determine whether a name that you want to give to a new definition will conflict with a predefined name.

---

†FORTH-79 Standard

The word FIND attempts to find the next word in the input stream in the dictionary and then returns its address or, if not found, a zero.

‡For polyFORTH Users

The word [-'] attempts to find the next word in the input stream in the dictionary. If the search is successful, [-'] leaves the parameter field address and false; if unsuccessful, leaves HERE and true.

You can also use the address to DUMP the contents of the definition, like this:

```
' GREET 12 DUMP
```

Or you can change the value of a constant by first finding its address, then storing the new value into it, like this:

```
110 ' LIMIT !
```

Or you can use tick to implement something called "vectored execution." Which brings us to the next section ...

Vectored Execution

While it sounds hairy, the idea of vectored execution is really quite simple. Instead of executing a definition directly, as we did with the phrase

```
' GREET EXECUTE
```

we can execute it indirectly by keeping its address in a variable, then executing the contents of the variable, like this:

```
' GREET POINTER !
POINTER @ EXECUTE
```

The advantage is that we can change the pointer later, so that a single word can be made to perform different things at different times.

Here is an example that you can try yourself:

```
1 : HELLO   ." HELLO " ;
2 : GOODBYE   ." GOODBYE " ;
3 VARIABLE 'ALOHA
4 : ALOHA   'ALOHA @ EXECUTE ;
5
6 ' HELLO  'ALOHA !
```

In the first two lines, we've simply created words which print the strings "HELLO" and "GOODBYE." In line 3, we've defined a variable called 'ALOHA. This will be our pointer. In line 4, we've defined the word ALOHA to execute the definition whose address is in 'ALOHA. In line 6, we store the address of HELLO into 'ALOHA.

Now if we execute ALOHA, we will get

```
ALOHA HELLO ok
```

Alternatively, if we execute the phrase

```
' GOODBYE 'ALOHA !
```

—to store the address of GOODBYE into 'ALOHA, we will get

```
ALOHA GOODBYE ok
```

—Thus the same word, ALOHA, can do two different things.

Notice that we named our pointer 'ALOHA (which we would pronounce tick-aloha). Since tick provides an address, we use it as a prefix to suggest "the address of" ALOHA. It is a FORTH naming convention to use this prefix for vectored execution pointers.

Tick always goes to the next word in the input stream.† What if we put tick inside a definition? When we execute the definition, tick will find the next word in the input stream, not the next word in the definition. Thus we could define

```
: SAY   ' 'ALOHA ! ;
```

then enter

```
SAY HELLO ok
ALOHA HELLO ok
```

or

```
SAY GOODBYE ok
ALOHA GOODBYE ok
```

to store the address of either HELLO or GOODBYE into 'ALOHA.

But what if we want tick to use the next word in the definition? We must use the word [']‡ (bracket-tick-bracket) instead of tick. For example:

```
: COMING   ['] HELLO  'ALOHA ! ;
: GOING    ['] GOODBYE 'ALOHA ! ;
```

---

†FORTH-79 Standard

The behavior of tick as described by the Standard differs somewhat from that explained here. See Appendix 3.

‡For Some Small-system, Non-polyFORTH, Users

If your keyboard doesn't have a "[" or "]" key, the documentation that came with your FORTH system should indicate substitutes.

Now we can say

```
COMING ok
ALOHA HELLO ok
GOING ok
ALOHA GOODBYE ok
```

Here's an example of vectored execution that can be found on certain FORTH systems. When FORTH is first loaded, the word NUMBER can only convert single-length numbers. But after double-length routines are loaded, NUMBER can convert double-length or single-length numbers. It would not be enough to simply redefine NUMBER, because then you would also have to redefine INTERPRET and any other word which uses NUMBER. Instead, the definition of NUMBER is something like

```
: NUMBER   'NUMBER @ EXECUTE ;
```

where 'NUMBER is the variable used as a pointer. When FORTH is first loaded, this variable contains the address of the single-length version. But when the double-length routines are loaded, a new definition called (NUMBER), with double-length capability, is added to the dictionary. On the line below the definition in the load block is the phrase

```
' (NUMBER) 'NUMBER !
```

When NUMBER is executed in the future, whether by INTERPRET or whomever, the contents of 'NUMBER are fetched and this definition is executed, giving NUMBER new-found double-length capability.

Here are the commands we've covered so far:†

' xxx	(-- adr)	Attempts to find the address of xxx (the word that follows in the input stream) in the dictionary.
[']	compile time: (--) run time: (-- adr)	Used only in a colon definition, compiles the address of the next word in the definition as a literal.

†FORTH-79 Standard

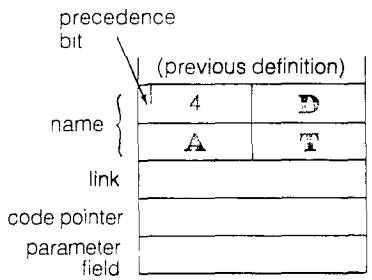
See Appendix 3.

The Structure of a Dictionary Entry

All definitions, whether they have been defined by :, by VARIABLE, by CREATE, or by any other "defining word," share these basic parts:

name field
link field
code pointer field
parameter field

Using the variable DATE as an example, here's how these components are arranged within each dictionary entry in systems that have a three-character-maximum name field. In this diagram, each horizontal line represents one cell in the dictionary:



Systems that allow thirty-one-character-maximum name fields usually follow the same pattern, but the name field may take anywhere from two to thirty-two bytes, depending on the name. The order of the four components may also vary.†

---

†FORTH-79 Standard

The FORTH-79 Standard allows thirty-one-character-maximum name fields, but does not specify the order of the field within the dictionary entry. The order is considered implementation-dependent.

In this book, we're only concerned with the functions of the four components, not with their order inside a dictionary entry. We'll use the three-character version as our example because it's the simplest.

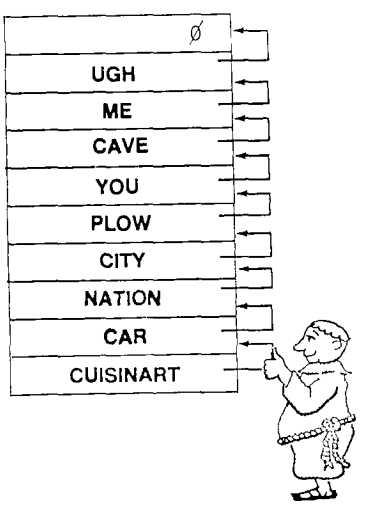
Name

In our example, the first byte contains the number of characters in the full name of the defined word (there are four letters in DATE). The next three bytes contain the ASCII representations of the first three letters in the name of the defined word. In a three-character system, this is all the information that tick or bracket-tick-bracket have to go on in matching up the name of a definition with a word in the input stream.

(Notice in the diagram that the sign bit of the "count" byte is called the "precedence bit." This bit is used during compilation to indicate whether the word is supposed to be executed during compilation, or to simply be compiled into the new definition. More on this matter in Chap. 11.)

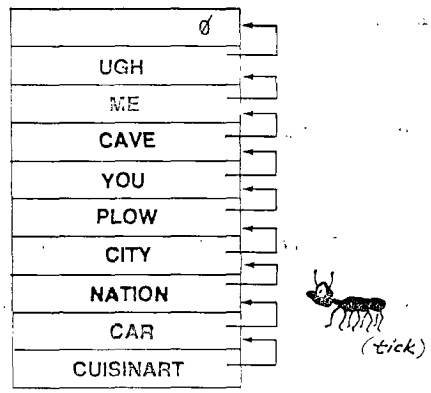
Link

The "link" cell contains the address of the previous definition in the dictionary list. The link cell is used in searching the dictionary. To simplify things a bit, imagine that it works this way:



Each time the compiler adds a new word to the dictionary, he sets the link field to point to the address of the previous definition. Here he is setting the link field of CUISINART to point to the definition of CAR.

At search time, tick (or bracket-tick-bracket, etc.) starts with the most recent word and follows the "chain" backwards, using the address in each link cell to locate the next definition back.

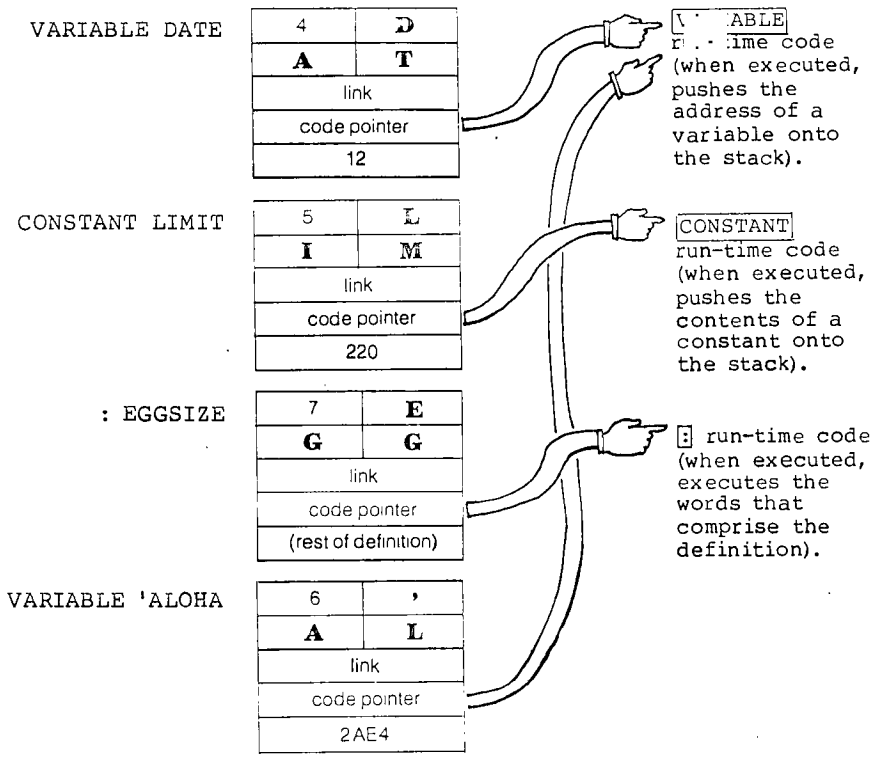


The link field of the first definition in the dictionary contains a zero, which tells tick to give up; the word is not in the dictionary.

Code pointer

Next is the "code pointer." The address contained in this pointer is what distinguishes a variable from a constant or a colon definition. It is the address of the instruction that is executed first when the particular type of word is executed. For example, in the case of a variable, the pointer points to code that pushes the address of the variable onto the stack. In the case of a constant, the pointer points to code that pushes the contents of the constant onto the stack. In the case of a colon definition, the pointer points to code that executes the rest of the words in the colon definition.

The code that is pointed to is called the "run-time code" because it's used when a word of that type is executed (not when a word of that type is defined or compiled).

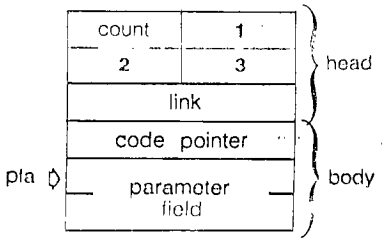


All variables have the same code pointer; all constants have the same code pointer of their own, and so on.

Parameter field

Following the code pointer is the parameter field. In variables and constants, the parameter field is only one cell. In a 2CONSTANT or 2VARIABLE, the parameter field is two cells. In an array, the parameter field can be as long as you want it. In a colon definition, the length of the parameter field depends on the length of the definition, as we'll explain in the next section.

The address that is supplied by tick and expected by EXECUTE is the address of the beginning of the parameter field, called the parameter-field address (pfa).



By the way, the name and link fields are often called the "head" of the entry; the code pointer and parameter fields are called the "body."

The Basic Structure of a Colon Definition

While the format of the head and code pointer is the same for all types of definitions, the format of the parameter field varies from type to type. Let's look at the parameter field of a colon definition.

The parameter field of a colon definition contains the addresses† of the previously defined words which comprise the definition. Here is the dictionary entry for the definition of PHOTOGRAPH, which we defined as:

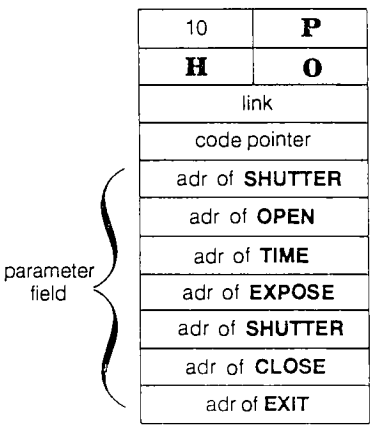
```
: PHOTOGRAPH  SHUTTER OPEN  TIME EXPOSE  SHUTTER CLOSE ;
```

When PHOTOGRAPH is executed, the definitions that are located at the successive addresses are executed in turn. The mechanism which reads the list of addresses and executes the definitions at each address is called the "address interpreter."

---

†For Experts

The addresses that comprise the body of a colon definition are usually code-field addresses (cfa), not parameter-field addresses.



The word [;] at the end of the definition compiles the address of a word called [EXIT]. As you can see in the figure, the address of [EXIT] resides in the last cell of the dictionary entry. The address interpreter will execute [EXIT] when it gets to this address, just as it executes the other words in the definition. [EXIT] terminates execution of the address interpreter, as we will see in the next section.

Nested Levels of Execution

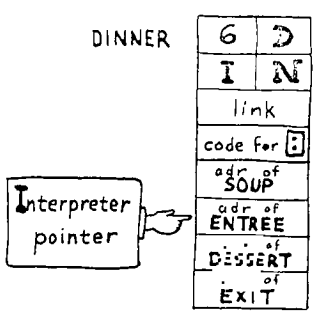
The function of [EXIT] is to return the flow of execution to the next higher-level definition that refers to the current definition. Let's see how this works in simplified terms.

Suppose that DINNER consists of three courses:

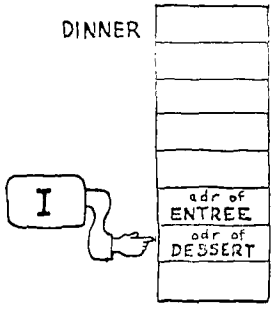
```
: DINNER   SOUP ENTREE DESSERT ;
```

and that tonight's ENTREE consists simply of

```
: ENTREE   CHICKEN RICE ;
```



We are executing DINNER and we have just finished the SOUP. The pointer that is used by the address interpreter is called the "interpreter pointer" ([I]). Since the next course after SOUP is the ENTREE, our interpreter pointer is pointing to the cell that contains the address of ENTREE.

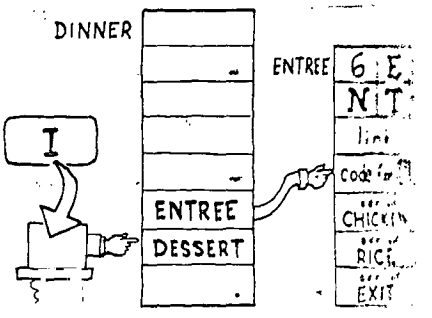


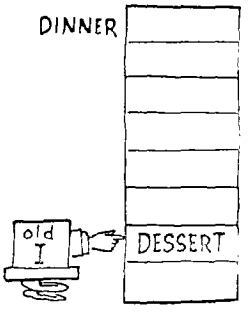
Before we go off and execute ENTR
we first increment the interpr
pointer so that when we come bac
will be pointing to DESSERT.

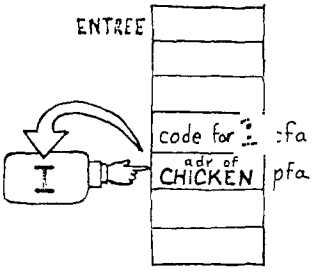
Now we begin to execute ENTREE. The first thing we execute is ENTREE's "code," i.e., the code that is pointed to by the "code field," common to all colon definitions.

This code does two things:

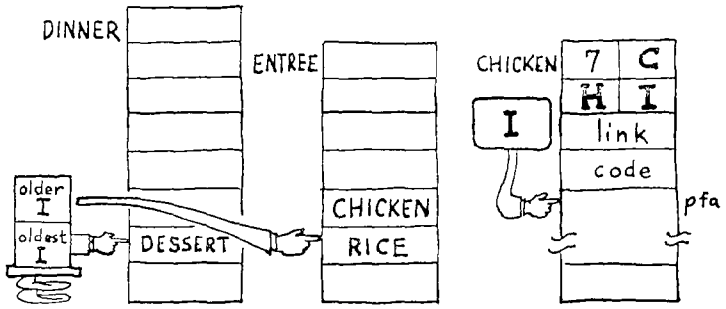
First, it saves the contents of the interpreter pointer on the return stack ...



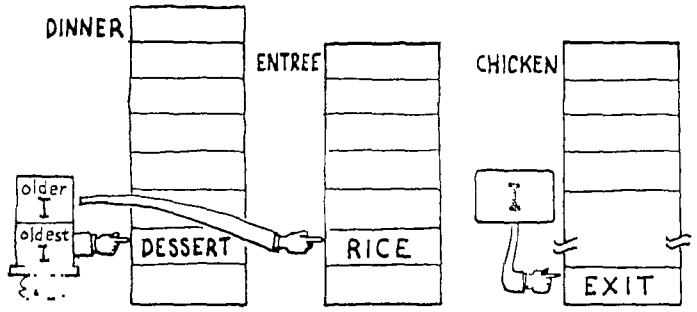




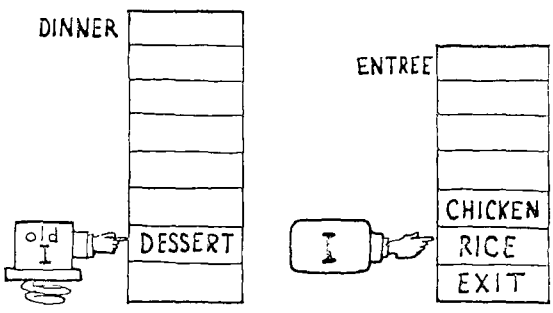
... then it puts
address of its
parameter fi
address (pfa) in
interpreter poin
Now the interpre
pointer is poin
to CHICKEN. So
address interp
gets ready to se
up the chicken.



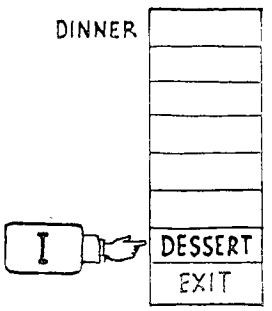
But first, as we did with ENTREE, we increment the pointer so that when we return it will be pointing to RICE. Then CHICKEN's code saves this pointer on the return stack and puts CHICKEN's own pfa into the interpreter pointer.



Finally we have our chicken, as the above process contiunes all down the line to the lowest-level definition involved in the ing of the succulent poultry. Sooner or later we come to the EXIT in CHICKEN.

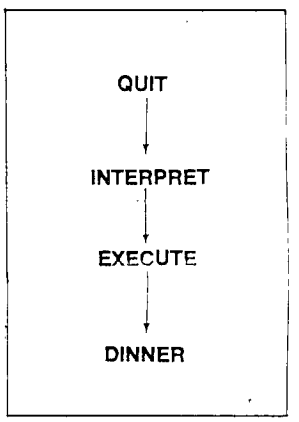


EXIT takes the number off the top of the return stack and puts it in the interpreter pointer. Now the address interpreter continues with the execution of RICE.



Eventually, of course, the EXIT in ENTREE will put the value on the return stack into the interpreter pointer. At last we're ready for DESSERT.

One Step Beyond



Perhaps you're wondering: what happens when we finally execute the EXIT in DINNER? Whose return address is on the stack? What do we return to?

Well, remember that DINNER has just been executed by EXECUTE, which is a component of INTERPRET. INTERPRET is a loop which checks the entire input stream. Assuming that we entered RETURN after DINNER, then there is nothing more to interpret. So when we exit INTERPRET, where does that leave us? In the outermost definition for each terminal, called QUIT.

QUIT, in simplified form, looks like this:

```
: QUIT   BEGIN  (clear return stack)  (accept input)
          INTERPRET ." ok" CR  0 UNTIL ;
```

(The parenthetical comments represent words and phrases not yet covered.) We can see that after the word INTERPRET comes a dot-quote message, "ok," and a CR, which of course are what we see after interpretation has been completed.

Next is the phrase

```
0 UNTIL
```

which unconditionally returns us to the beginning of the loop, where we clear the return stack and once again wait for input.

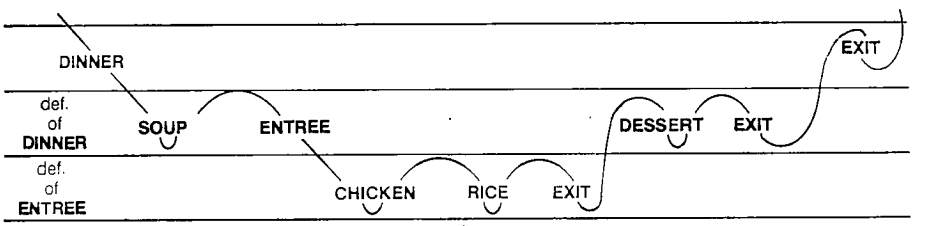
If we execute QUIT at any level of execution, we will

immediately cease execution of our application and re-enter QUIT's loop. The return stack will be cleared (regardless of how many levels of return addresses we had there, since we could never use any of them now), and the system will wait for input. You can see why QUIT can be used to keep the message "ok" from appearing at our terminal.

The definition of ABORT" uses QUIT.

Abandoning The Nest

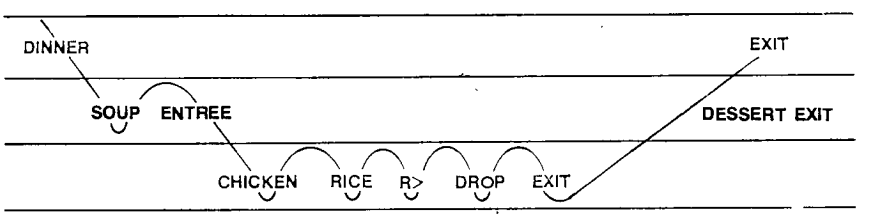
It's possible to skip one level of execution simply by removing one return address from the return stack. For example, consider the three levels of execution associated with DINNER, shown here:



Now suppose that the definition ENTREE is changed to:

```
: ENTREE   CHICKEN RICE  R> DROP ;
```

The phrase "R> DROP" will drop from the return stack the return address of DESSERT, which was put on just prior to the execution of ENTREE. If we reload these definitions and execute DINNER, the EXIT on the third level will take us directly back to the first level. We'll get SOUP, CHICKEN, and RICE but we'll skip DESSERT, as you can see here:

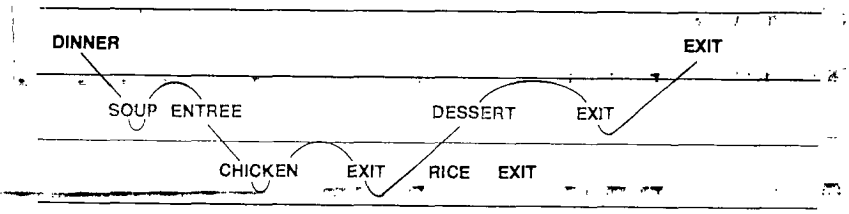


We're not necessarily suggesting that you use "R> DROP" in an application, just illustrating a point.

We've mentioned that the word EXIT removes a return address from atop the return stack and puts it into the interpreter pointer. The address interpreter, which gets its bearings from the interpreter pointer, begins looking at the next level up. It's possible to include EXIT in the middle of a definition. For example, if we were to redefine ENTREE as follows:

```
: ENTREE   CHICKEN  EXIT  RICE ;
```

then when we subsequently execute DINNER, we will exit right after CHICKEN and return to the next course after the ENTREE, i.e., DESSERT.

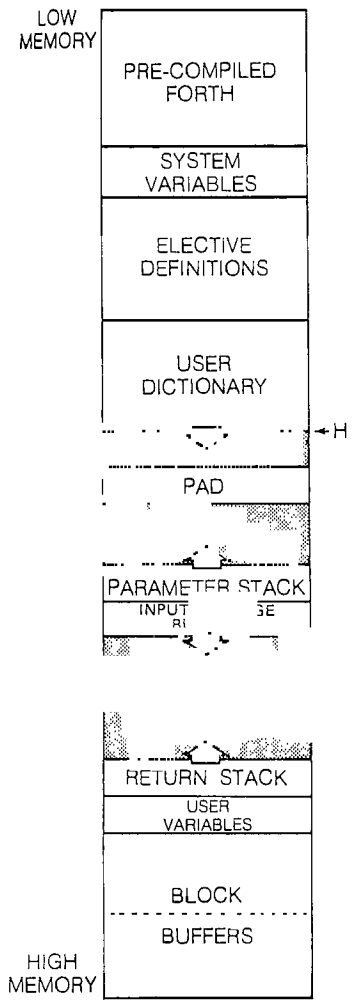


This time we get DESSERT but no RICE.

EXIT is commonly used in a disk block to keep the remainder of the block from being loaded. For example, if you edit EXIT into the end of line 5 of a block and load it, any definitions in line 6 and beyond will not get compiled.

EXIT	(--)	When compiled within a colon definition, terminates execution of that definition at that point. When executed from a load block, terminates interpretation of the block at that point.
QUIT	(--)	Clears both stacks and returns control to the terminal. No message is given.

FORTH Geography



This is a "memory map"† of a typical single-user FORTH system. Multiprogrammed systems such as polyFORTH are more complicated, as we will explain later on. For now let's take the simple case and explore each region of the map, one at a time.

Precompiled Portion

In low memory resides the only precompiled portion of the system (already compiled into dictionary form). On some systems this code is kept on disk (often blocks 1 - 8) and automatically loaded into low RAM when you start up or "boot" the computer. On other systems the precompiled portion resides permanently in PROM, where it is active as soon as you power up the computer.

The precompiled portion usually includes most of the single-length math operators and number-formatting words, single-length stack manipulation operators, editor commands, branching and structure-control words, the assembler, all the defining words we've covered so

---

†For Beginners

A "memory map" depicts how computer memory is divided up for various purposes in a particular system. Here, low-numbered addresses begin at the top ("low memory") and increase as the map goes down. Memory space is measured in groups of 1,024 bytes. This quantity is called a "K" (from "kilo-," meaning a thousand, which is close enough).

far, and, of course, the text and address interpreters.†

System Variables

The next section of memory contains "system variables" which are created by the precompiled portion and used by the entire system. They are not generally used by the user. 'NUMBER, which we discussed earlier, is a system variable.

Elective Definitions

The portion of the FORTH system that is not precompiled is kept on disk in source-text form. You can elect to load or not to load any number of these definitions to better control use of your computer's memory space. The load block for all "electives" is called the "electives block," usually block 9. To compile the electives after you "boot," simply enter

```
9 LOAD
```

(or whichever block is the electives block for your system).

For example, in polyFORTH electives include double- and mixed-length operators, extended editor commands, date and time commands, and the ability to add new multiprogrammed tasks including additional terminals. You can mask any of these electives out of the electives block simply by inserting parentheses.

If your electives block contains this line:

```
( 32-BIT ARITHMETIC)  30 LOAD   31 LOAD   32 LOAD
```

you can avoid loading the double-length routines by changing the line to

```
( 32-BIT ARITHMETIC  30 LOAD   31 LOAD   32 LOAD)
```

If you want to change the electives block after you have already loaded it, you must reload the system (by rebooting) before you can reload the electives. (The word [illegible]LOAD, available on some systems, will reload the system and not the electives.)

---

† For Experts

To give you an idea of how compact FORTH can be, all of polyFORTH's precompiled portion resides in less than 8K bytes.

User Dictionary

The dictionary will grow into higher memory as you add your own definitions within the portion of memory called the "user dictionary." The next available cell in the dictionary at any time is pointed to by a variable called H. During the process of compilation, the pointer H is adjusted cell-by-cell (or byte-by-byte) as the entry is being added to the dictionary. Thus H is the compiler's bookmark; it points to the place in the dictionary where the compiler can next compile.

H is also used by the word ALLOT, which advances H by the number of bytes given. For example, the phrase

```
10 ALLOT
```

adds ten to H so that the compiler will leave room in the dictionary for a ten-byte (or five-cell) array.

A related word is HERE, which is simply defined

```
: HERE  H @ ;
```

to put the value of H on the stack. The word , (comma), which stores a single-length value into the next available cell in the dictionary, is simply defined

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

that is, it stores a value into HERE and advances the dictionary pointer two bytes to leave room for it.

You can use H . S to determine how much memory any part of your application requires, simply by comparing the HERE from before with the HERE after compiling. For example:

```
HERE  220 LOAD  HERE SWAP - . 196 ok
```

indicates that the definitions loaded by block 220 filled 196 bytes of memory space in the dictionary.

The Pad

At a certain distance from HERE in your dictionary, you will find a small region of memory called the "pad." Like a scratch pad, it is usually used to hold ASCII character strings that are being manipulated prior to being sent out to a terminal. For example, the number-formatting words use the pad to hold the ASCII numerals during the conversion process, prior to TYPE.

The size of the pad is indefinite. In most systems there are hundreds or even thousands of bytes between the beginning of the pad and the top of the parameter stack.

Since the pad's beginning address is defined relative to the last dictionary entry, it moves every time you add a new definition or execute FORGET or EMPTY. This arrangement proves safe, however, because the pad is never used when any of these events are occurring. The word PAD returns the current address of the beginning of the pad. It is defined simply:

```
: PAD   HERE  34 + ;
```

that is, PAD returns an address that is a fixed number of bytes beyond HERE. (The actual number may vary.)

Parameter Stack

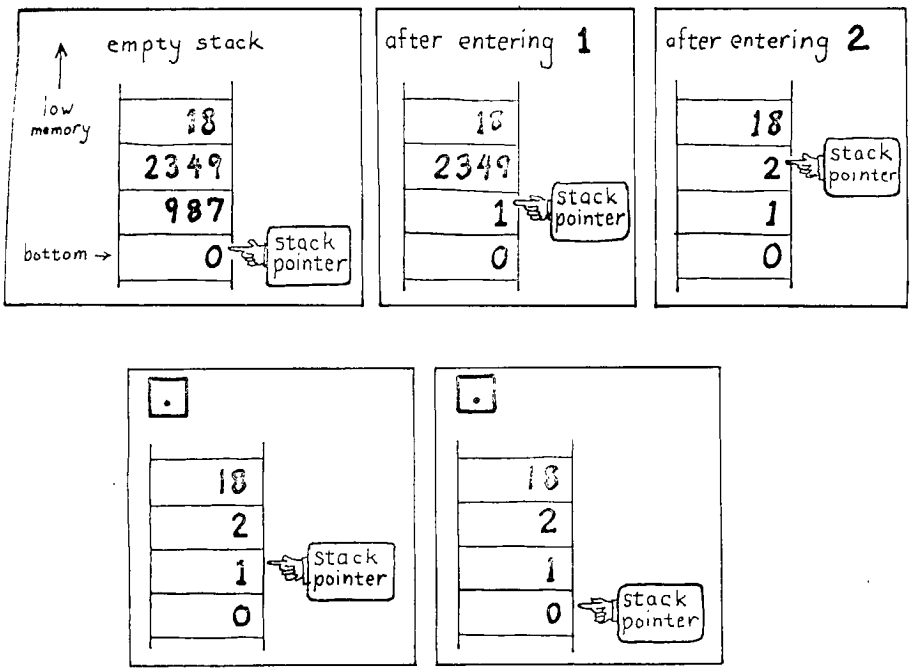
Far above† the pad in memory is the area reserved for the parameter stack. Although we like to imagine that values actually move up and down somewhere as we "pop them off" and "push them on," in reality nothing moves. The only thing that changes is a pointer to the "top" of the stack.

As you can see below, when we "put a number on the stack," what really happens is that the pointer is "decremented" (so that it points to the next location toward low memory), then our number is stored where the pointer is pointing. When we "remove a number from the stack," the number is fetched from the location where the pointer is pointing, then the pointer is incremented. Any numbers above the stack pointer on our map are meaningless.

---

†For Beginners

"Above" refers to the higher memory addresses, which are "lower" on our map.



As new values are added to the stack, it "grows toward low memory."

The stack pointer is fetched by the word 'S (pronounced tick-S). Since 'S provides the address of the top stack location, the phrase

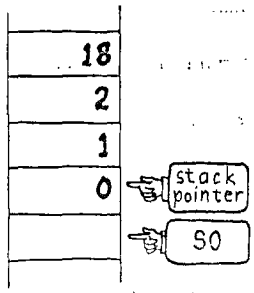
```
'S @
```

fetches the contents of the top of the stack. This operation, of course, is identical to that of DUP. If we had five values on the stack, we could copy the fifth one down with the phrase

```
'S 8 + @
```

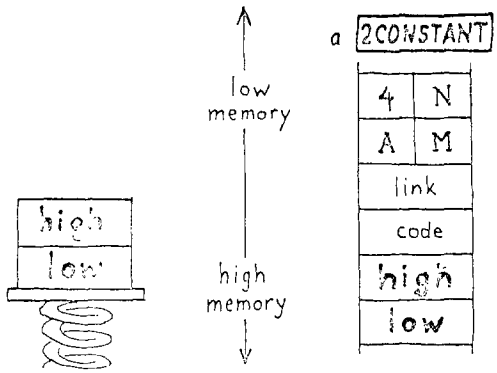
(but this is generally not considered good programming practice).

The bottom of the stack is pointed to by a variable called S0 (S-zero). S0 always contains the address of the next cell below the "empty stack" cell.



For examples of good uses of 'S and S0, review the definitions of DEPTH and of .S that we gave in the Handy Hint at the end of Chap. 3.

Notice that with double-length numbers, the high-order cell is stored at the lower memory address whether on the stack or in the dictionary. The operators 2! and 2@ keep the order of cells consistent, as you can see here.



Input Message Buffer

S0 also contains the starting address for the "input message buffer," which grows toward high memory (the same direction as the pad). When you enter text from the terminal, it gets stored into this buffer where the text interpreter will scan it.

Return Stack

Above the buffer resides the return stack, which operates identically to the parameter stack. There are no high-level FORTH words analogous to 'S or S0 that refer to the return stack.

User Variables

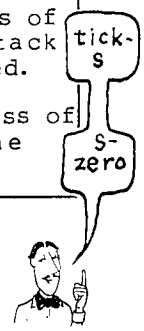
The next section of memory contains "user variables." These variables include H, S0, and many others that we'll cover in an upcoming section.

Block Buffers

At the high end of memory reside the block buffers. Each buffer provides 1,024 bytes for the contents of a disk block. Whenever you access a block (by listing or loading it, for example) the system copies the block from the disk into the buffer, where it can be modified by the editor or interpreted by LOAD. We'll discuss the block buffers in Chap. 10.

This completes our journey across the memory map of a typical single-user FORTH system. Here are the words we've just covered that relate to memory regions in the FORTH system.†

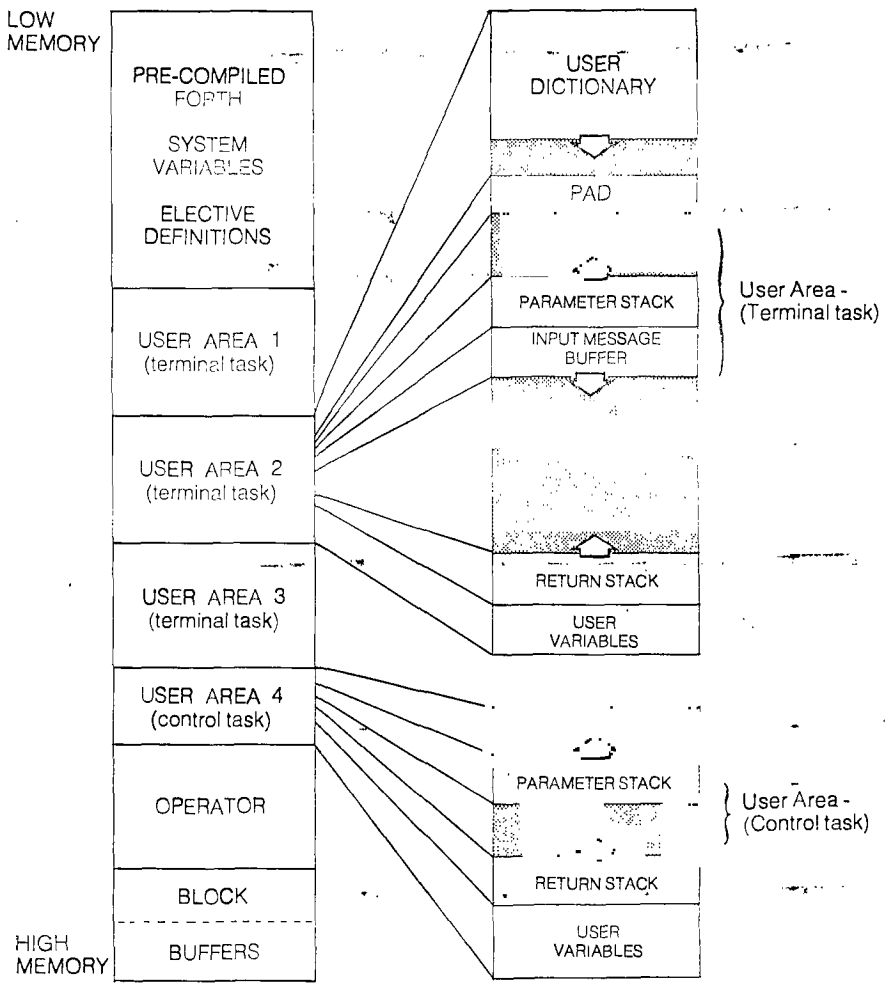
H	(-- adr)	Returns the address of the dictionary pointer.
HERE	(-- adr)	Returns the next available dictionary location.
PAD	(-- adr)	Returns the beginning address of a scratch area used to hold character strings for intermediate processing.
'S	(-- adr)	Returns the address of the top of the stack before 'S is executed.
S0	(--adr)	Contains the address of the bottom of the parameter stack.



†FORTH-79 Standard

H, 'S, and S0 are not required by the Standard.

The Geography of a Multi-tasked FORTH system



Some FORTH systems (such as polyFORTH) can be multitasked,† so that any number of additional tasks can be added. A task may be

†For Beginners

The term "multitasked" describes a system in which numerous tasks operate concurrently on the same computer without interference from one another.

either a "terminal task," which puts the full interactive power of FORTH into the hands of a human at a terminal, or a "control task," which controls a hardware device that has no terminal.

Either type of task requires its own "user area." The size and contents of a user area depends on the type of task, but typical configurations for the two types of tasks are shown in the figure.

Each terminal task has its own private dictionary, pad, parameter stack, input message buffer, return stack, and user variables. This means that any words that you define at your terminal are normally not available to other terminals. Similarly, each task has its own copies of the user variables, such as BASE.

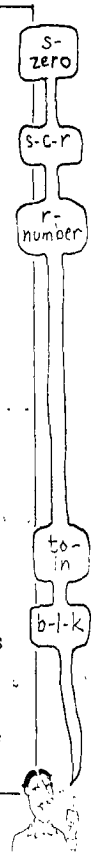
Each control task has a pair of stacks and a small set of user variables. Since a control task uses no terminal, it doesn't need a dictionary of its own; nor does it need a pad or a message buffer.

Following the initial boot there is only one task, called OPERATOR. Loading the electives block will allocate space for the various terminal and control-task partitions. Thus it is possible to reconfigure the subtasks within a system by altering the electives block and reloading it. But it's beyond the scope of this book to explain how.

User Variables

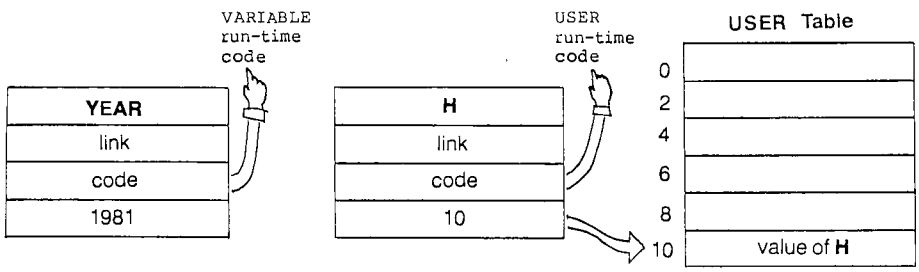
The following list shows most of the user variables. Some we won't ever mention again. Don't try to memorize this table. Just remember where you can find it.

S0	Pointer to the bottom of the parameter stack and, for terminal tasks, the start of the input message buffer.
SCR	For the editor, a pointer to the current block number (set by LIST and used by L).
R#	Current character position in the editor.
BASE	Number conversion base.
H	Dictionary pointer. Pointer to the next available byte.
CONTEXT	Contains up to four indexes for vocabularies to be searched.
CURRENT	Contains the index of the vocabulary to which new definitions will be linked.
>IN	Pointer to the current position in the input stream.
BLK	If non-zero, a pointer to the block being interpreted by LOAD. A zero indicates interpretation from the terminal (via the input message buffer).
OFFSET	Block offset to disk drives. The content of OFFSET is added to the stack number by BLOCK.



User variables are not like ordinary variables. With an ordinary variable (one defined by the word VARIABLE), the value is kept in the parameter field of the dictionary entry.

Each user variable, on the other hand, is kept in an array called the "user table." The dictionary entry for each user variable is located elsewhere; it contains an offset into the user table. When you execute the name of a user variable, such as H, this offset is added to the beginning address of the user table. This gives you the address of H in the array, allowing you to use @ or ! in the normal way.



The main advantage of user variables is that any number of tasks can use the same definition of a variable and each get its own value. Each task that executes

```
BASE @
```

gets the value for BASE from its own user table. This saves a lot of room in the system while still allowing each task to execute independently.

User variables are defined by the word USER. The sequence of user variables in the table and their offset values vary from one system to another.

To summarize, there are three kinds of variables: System variables contain values used by the entire FORTH system. User variables contain values that are unique for each task, even though the definitions can be used by all tasks in the system. Regular variables can be accessible either system-wide or within a single task only, depending upon whether they are defined within OPERATOR or within a private task.

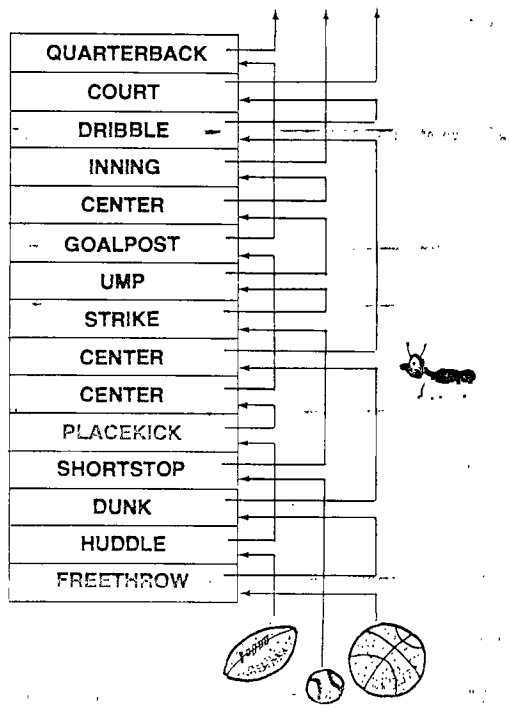
Vocabularies

Earlier we mentioned that the reason the I in the editor doesn't conflict with the I used in a DO loop is that they belong to separate "vocabularies." In a simple FORTH system there are three standard vocabularies: FORTH, the editor, and the assembler.

All the words that we've covered so far belong to the FORTH vocabulary, except for the editor commands which belong to the editor vocabulary. The assembler vocabulary contains commands that are used to write assembly-language code for your particular computer. Since assembly code varies from computer to computer, and since assembly-language programming is a whole different subject, we won't cover it in this book.†

All definitions are added to the same dictionary in the order in which they are compiled, regardless of which vocabulary they belong to. So vocabularies are not subdivisions of the dictionary; rather they are independently linked lists that weave through it.

For example, in the figure shown here, there are three vocabularies: football, baseball, and basketball. All three are co-resident in the same dictionary, but when tick follows the basketball chain, for instance, it only finds words in the basketball vocabulary. Even though each vocabulary has a word called CENTER, tick will find whichever version is appropriate for the context.



†For the Curious

See Appendix 2.

There is another advantage besides exclusivity, and that is speed of searches. If we are talking about basketball, why waste time hunting through the football and baseball words?

You can change the context in which the dictionary is searched by executing any of the three commands FORTH, EDITOR, or ASSEMBLER. For example, if you enter

```
FORTH
```

you know for sure that the search context is the FORTH vocabulary.

Ordinarily, however, the FORTH system automatically changes the context for you. Here's a typical scenario:

The system starts out with FORTH being the context. Let's say you start entering an application into a block. Certain editor commands switch the context to the editor vocabulary. You will stay in the editor vocabulary until you load the block and begin compiling definitions. The word ; will automatically reset the context to what it was before--FORTH.

Different versions of FORTH have different ways of implementing vocabularies. Still, we can make a few general statements that will cover most systems.

The vocabulary to be searched is specified by a user variable called CONTEXT. As we said, the commands FORTH, EDITOR, and ASSEMBLER change the search context.

There is another kind of vocabulary "context": the vocabulary to which new definitions will be linked. The link vocabulary is specified by another variable called CURRENT. Because CURRENT normally specifies the FORTH vocabulary, new definitions are normally linked to the FORTH vocabulary.

But how does the system compile words into the editor and assembler vocabularies? By using the word DEFINITIONS, as in

```
EDITOR DEFINITIONS
```

We know that the word EDITOR sets CONTEXT to "EDITOR." The word DEFINITIONS copies whatever is in CONTEXT into CURRENT. The definition of DEFINITIONS is simply

```
: DEFINITIONS   CONTEXT @  CURRENT ! ;
```

Having entered

```
EDITOR DEFINITIONS
```

any words that you compile henceforth will belong to the editor

vocabulary until you enter

FORTH DEFINITIONS

to reset CURRENT to "FORTH."†

We've presented this introduction to vocabularies mainly to satisfy your curiosity, not to encourage you to add new vocabularies of your own. The problem of defining different subsets of application words with conflicting names is better handled by the use of overlays, which we discussed in Chap. 3.

†For Curious polyFORTH Users

polyFORTH allows several vocabularies to be chained in sequence. CONTEXT specifies the search order.

The polyFORTH dictionary is comprised of eight "linked lists" which do not correspond with the vocabularies. At compile time a hashing function, based on (usually) the first letter of the word being defined, computes a "hashing index." This index is combined with the "current" vocabulary to produce an index into one of the eight lists.

Thus a single list may contain words from many vocabularies, but any words with identical names belonging to separate vocabularies will be linked to separate lists. The distribution of entries in each chain is balanced, and an entire vocabulary can be searched by searching only one-eighth of the dictionary.

A Handy Hint

How to LOCATE a Source Definition

Some FORTH systems, such as polyFORTH, feature a very useful word called LOCATE. If you enter

```
LOCATE EGGSIZE
```

FORTH will list the block that contains the definition of EGGSIZE. The only requirements are that the word must be resident (currently in the dictionary) and that the word must have been loaded from a block. You therefore can locate system electives and words in your application, but you can't locate words in the precompiled portion.

' xxx	(-- adr)	Attempts to find the address of xxx (the word that follows in the input stream) in the dictionary.
INTERPRET	(--)	Interprets the input stream, indexed by >IN , until exhausted.
EXECUTE	(adr --)	Executes the dictionary entry whose parameter field address is on the stack.
EXIT	(--)	When compiled within a colon definition, terminates execution of that definition at that point. When executed from a load block, terminates interpretation of the block at that point.
QUIT	(--)	Clears both stacks and returns control to the terminal. No message is given.
HERE	(-- adr)	Returns the next available dictionary location.
PAD	(-- adr)	Returns the beginning address of a scratch area used to hold character strings for intermediate processing.
FORTH	(--)	Makes FORTH the CONTEXT vocabulary.
EDITOR	(--)	Makes the editor vocabulary the CONTEXT vocabulary.
ASSEMBLER	(--)	Makes the assembler vocabulary the CONTEXT vocabulary.
DEFINITIONS	(--)	Sets CURRENT to the CONTEXT vocabulary so that subsequent definitions will be linked to this vocabulary.

Common User Variables
(Some not required by the FORTH-79 Standard.)

S0	Pointer to the bottom of the parameter stack and, for terminal tasks, the start of the input message buffer.
SCR	For the editor, a pointer to the current block number (set by LIST and used by L).
R#	Current character position in the editor.
BASE	Number conversion base.
H	Dictionary pointer. Pointer to the next available byte.
CONTEXT	Contains up to four indexes for vocabularies to be searched.
CURRENT	Contains the index of the vocabulary to which new definitions will be linked.
>IN	Pointer to the current position in the input stream.
BLK	If non-zero, a pointer to the block being interpreted by LOAD. A zero indicates interpretation from the terminal (via the input message buffer).
OFFSET	Block offset to disk drives. The content of OFFSET is added to the stack number by BLOCK.

Additional Words Available in Some Systems

[']	compile time: (--) run time: (-- adr)	Used only in a colon definition, compiles the address of the next word in the definition as a literal.
'S	(-- adr)	Returns the address of the top of the stack before 'S is executed.

Review of Terms

Address interpreter	the second of FORTH's two interpreters, the one which executes the list of addresses found in the dictionary entry of a colon definition. The address interpreter also handles the nesting of execution levels for words within words.
Body	the code and parameter fields of a FORTH dictionary entry.
Boot	simply, to load the precompiled portion of FORTH into the computer so that you can talk to the computer in FORTH. This happens automatically when you turn the computer on or press "Reset."
Cfa	code field address; the address of a dictionary entry's code pointer field.
Control task	on a multitasked system, a task which cannot converse with a terminal. Control tasks usually run hardware devices.
Code pointer field	the cell in a dictionary entry which contains the address of the run-time code for that particular type of definition. For example, in a dictionary entry created by :, the field points to the address interpreter.
Defining word	a FORTH word which creates a dictionary entry. Examples include :, CONSTANT, VARIABLE, etc.
Electives	the set of FORTH definitions that come with a system but not in the precompiled portion. The "electives block" loads the blocks that contain the elective definitions; the block can be modified as the user desires.
Head	the name and link fields of a FORTH dictionary entry.
Input message buffer	the region of memory within a terminal task that is used to store text as it arrives from a terminal. Incoming source text is interpreted here.

Link field	the cell in a dictionary entry which contains the address of the previous definition, used in searching the dictionary. (On systems which use multiple chains, the link field contains the address of the previous definition in the same chain.)
Name field	the area of a dictionary entry which contains the name (or abbreviation thereof) of the defined word, along with the number of characters in the name.
Pad	the region of memory within a terminal task that is used as a scratch area to hold character strings for intermediate processing.
Parameter field	the area of a dictionary entry which contains the "contents" of the definition: for a CONSTANT, the value of the constant; for a VARIABLE, the value of the variable; for a colon definition, the list of addresses of words that are to be executed in turn when the definition is executed. Depending on its use, the length of a parameter field varies.
Pfa	parameter field address; the address of the first cell in a dictionary entry's parameter field (or, if the parameter field consists of only one cell, its address).
Precompiled portion	the part of the FORTH system which is resident in object form immediately after the power-up or boot operation. The precompiled portion usually includes the text interpreter and the address interpreter; defining, branching, and structure-control words; single-length math and stack operators; single-length number conversion and formatting commands; the editor; and the assembler.
Run-time code	a routine, compiled in memory, which specifies what happens when a member of a given class of words is executed. The run-time code for a colon definition is the address interpreter; the run-time code for a variable pushes the contents of the variable's pfa onto the stack.
System variable	one of a set of variables provided by FORTH which are referred to system-wide (by any task). Contrast with "user variable."

Task	in FORTH, a partition in memory that contains at minimum a parameter and a return stack and a set of user variables.
Terminal task	on a multitasked system, a task which can converse with a human being using a terminal; i.e., one which has a text interpreter, dictionary, etc.
User variable	one of a set of variables provided by FORTH, whose values are unique for each task. Contrast with "system variable."
Vectored execution	the method of specifying code to be executed by providing not the address of the code itself but the address of a location which contains the address of the code. This location is often called the "vector." As circumstances change within the system, the vector can be reset to point to some other piece of code.
Vocabulary	an independently linked subset of the FORTH dictionary.

Problems -- Chapter 9

1. First review Chap. 2, Prob. 6. Without changing any of those definitions, now write a word called COUNTS which will allow the judge to optionally enter the number of counts for any crime. For instance, the entry

```
CONVICTED-OF BOOKMAKING 3 COUNTS TAX-EVASION
WILL-SERVE RETURN 17 YEARS ok
```

   will compute the sentence for one count of bookmaking and three counts of tax evasion.

2. What is the beginning address of your private dictionary?

3. In your system, how far is the pad from the top of your private dictionary?

4. Assuming that DATE has been defined by VARIABLE, what is the difference between these two phrases:

```
DATE .
```

   and

```
' DATE .
```

   What is the difference between these two phrases:

```
BASE .
```

   and

```
' BASE .
```

5. In this exercise you will create a "vectored execution array," that is, an array which contains addresses of FORTH words. You will also create an operation word which will execute one word stored in the array when the operation word is executed.

   Define a one-dimensional array of two-byte elements which will return the nth element's address when given a preceding subscript n. Define several words which output something at your terminal and take no inputs. Store the addresses of these output words in various elements of the array. Store the address of a do-nothing word in any remaining elements

of the array. Define a word which will take a valid array index and execute the word whose address is stored in the referenced element.

For example,

```
1 DO-SOMETHING HELLO, I SPEAK FORTH. ok
2 DO-SOMETHING 1 2 3 4 5 6 7 8 9 10 ok
3 DO-SOMETHING
**********
**********
**********
**********
**********
4 DO-SOMETHING ok
5 DO-SOMETHING ok
```