

10 I/O AND YOU

In this chapter we'll explain how FORTH handles I/O[†] of character strings to and from the block buffers and the terminal.

Specifically, we'll discuss disk-access commands, output commands, string-manipulation commands, input commands, and number-input conversion.

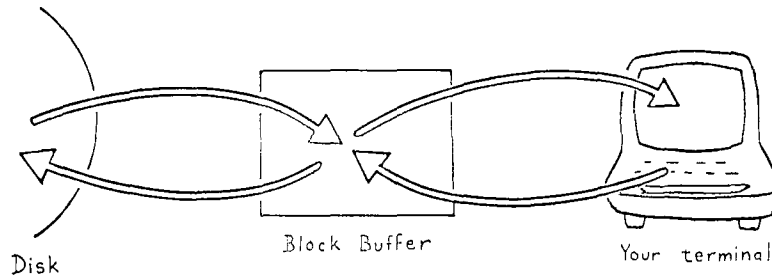
Block Buffer Basics

The FORTH system is designed so that you don't usually need to think about the mechanics of the block buffers. But sooner or later you will, so here's how it works.

As we mentioned earlier, each buffer is large enough to hold the contents of one block (1024 bytes) in RAM so that it can be edited, loaded, or generally accessed in any way. While we can imagine that we're communicating directly to the disk, in reality, the system brings the data from the disk into the buffer where we can read it. We can also write data to the buffer, and the system will send it along to the disk.

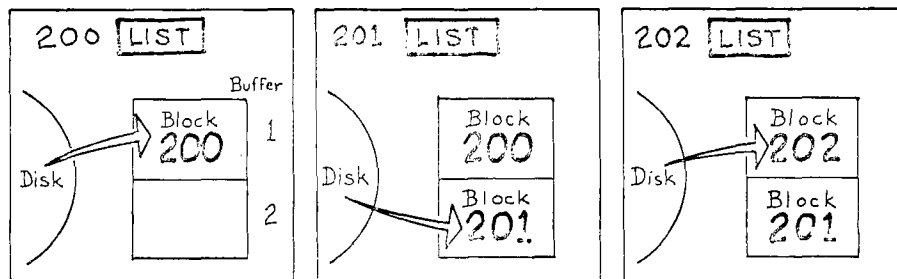
[†]For Beginners

I/O is an abbreviation for "input-output," which refers to data, text, or signals that are sent or received by the computer. I/O devices include terminals, printers, disk drives, push buttons, etc.



This arrangement is called "virtual memory" because the mass storage memory is made to act like computer memory.

Many FORTH systems use as few as two block buffers, even when the system is multiprogrammed. Let's see how this is possible.



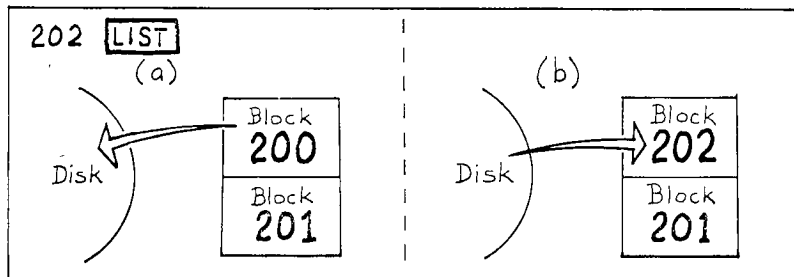
Suppose there are two buffers in your system. Now imagine the following scenario:

First you list block 200. The system reads the disk and transfers the block to buffer 1, from which `LIST` displays it.

Now you list block 201. The system copies block 201 from the disk into the other buffer.

Now you list block 202. The system copies block 202 from the disk into the less-recently used buffer, namely buffer 1.

What happened to the former contents of buffer 1? They were simply overwritten (erased) by the new contents. This is no loss because block 200 is still on the disk. But what if you had edited block 200? Would your changes be lost? No. Here's what would happen when you listed block 202:



First the modified contents of block 200 would be sent to the disk to update the former contents of 200 there, then the contents of 202 would be brought into the buffer.

The magic word is `UPDATE`, which sets a flag that indicates that the contents of the most recently accessed buffer should be sent back to disk, rather than erased, the next time that the buffer is needed. All editor commands that change the contents of a block, whether adding or deleting, include `UPDA` in their definitions.

Every time you or the system try to access a block, the system first checks whether the block is already in a buffer. If it is, fine. If not, then the system finds the earliest buffer to have been accessed. If the contents of this buffer have been `UPDATE`d, the system copies the contents back onto disk, then finally copies the newly-accessed block into the buffer.

This arrangement lets you modify the contents of the block any number of times without activating the disk drive each time. Since conversing with the disk takes longer than conversing with RAM, this can save a lot of time.

On the other hand, when there are several users on a single system, this arrangement allows all of them to get by with as few as two buffers (2K of memory), even though each may be accessing a different block.

Some FORTH systems give their owners the option to have as many block buffers as they like, depending on the memory size and the frequency of disk transfers in their own setups.

The word `FLUSH`[†] forces all updated buffers to be written to disk

[†]FORTH-79 Standard

The Standard's name for `FLUSH` is `SAVE-BUFFERS`.

immediately. Now that you know about the buffers, you can see why we need `FLUSH`: merely updating a buffer doesn't get it written to disk.

You should also know that when you `FLUSH`, the system "forgets" that it has your block in a buffer and clears the buffer's update flag. If you list or load the block again, FORTH will have to read it from the disk again.

The effective opposite of `FLUSH` is `EMPTY-BUFFERS`, which also makes the system "forget" any block it has and clears any update flags. `EMPTY-BUFFERS` is useful if you've accidentally got "garbage"† in a buffer (e.g., you've deleted some important lines and forgotten what you had originally, or generally messed up) and you don't want it to get forced onto the disk. When you list your block again, after entering `EMPTY-BUFFERS`, the system won't know it ever had your block in memory and will bring it in off the disk anew.‡

Each buffer has an associated cell in memory called the "buffer status cell." It contains the number of the block (e.g., 180).‡ The system uses it to tell whether a requested block is already in memory. When you `COPY` a block, all you are really doing is changing the number of the block in the buffer status cell and updating the buffer. When it's time for the buffer to be written to disk, it will be written to the new block.

The basic word that brings a block in from the disk, after first finding an available buffer and storing its contents on disk if necessary, is `BLOCK`. For instance, if you say

205 BLOCK

the system will copy block 205 from disk into one of the buffers. `BLOCK` also leaves on the stack the address of the beginning of the buffer that it used. We'll learn how to use this address in a few sections.

†For Beginners

"Garbage" is computer jargon for data which is wrong, meaningless, or irrelevant for the use to which it is being put.

‡For Those Using a Multiprogrammed System

Careful! `EMPTY-BUFFERS` empties everyone's buffers.

‡ For the Curious

The sign bit of the buffer status cell serves as the "update flag." If the number in the buffer status cell tests as negative by `0<`, then the buffer has been "updated."

If your application requires writing a lot of data to the disk without reading what's on the disk already (e.g., to initialize a disk, write raw data, transfer tape to disk, etc.), then you'll want to use `BUFI` . . .

`BUFFER` is used by `BLOCK` to assign a block number to the next available buffer. `BUFI` doesn't read the contents of the disk into the buffer. Also, `BUFI` doesn't check to see whether the block number has already been assigned to a buffer, so you have to make sure that no two buffers get assigned to the same number.

UPDATE	(--)	Marks the most recently referenced block as modified. The block will later be automatically transferred to mass storage if its buffer is needed to store a different block or if FLUSH is executed.
EMPTY-BUFFERS	(--)	Marks all block buffers as empty without necessarily affecting their actual contents. Updated blocks are not written to mass storage.
BLOCK	(u -- adr)	Leaves the address of the first byte in block u. If the block is not already in memory, it is transferred from mass storage into whichever memory buffer has been least recently accessed. If the block occupying that buffer has been updated (i.e., modified), it is rewritten onto mass storage before block u is read into the buffer.
BUFFER	(u -- adr)	Obtains the next block buffer, assigning it to block u. The block is not read from mass storage.

Output Operators

The word `EMIT` takes a single ASCII representation on the stack, using the low-order byte only, and prints the character at your terminal. For example, in decimal:

```
65 EMIT Aok
66 EMIT Bok
```

The word `TYPE` prints an entire string of characters at your terminal, given the starting address of the string in memory and the count, in this form:

```
(adr u -- )
```

We've already seen `TYPE` in our number-formatting definitions without worrying about the address and count, because they are automatically supplied by `#>`.

Let's give `TYPE` an address that we know contains a character string. Remember that the starting address of the input message buffer is kept by the user variable `S0`? Suppose we enter the following command:

```
S0 @ 12 TYPE
```

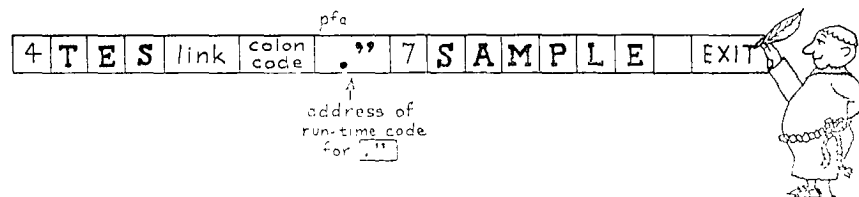
This will type twelve characters from the input message buffer, which contains the command we just entered:

```
S0 @ 12 TYPE return S0 @ 12 TYPEok
```

Let's digress for a moment to look at the operation of `."`. At compile time, when the compiler encounters a dot-quote, it compiles the ensuing string right into the dictionary, letter-by-letter, up to the delimiting double-quote. To keep track of things, it also compiles the count of characters into the dictionary entry. Given the definition

```
: TEST ." SAMPLE " ;
```

and looking at bytes in the dictionary horizontally rather than vertically, here is what the compiler has compiled:



if we wanted to, we could type the word "SAMPLE" ourselves (without executing TEST) with the phrase

```
' TEST 3 + 7 TYPE
```

where

```
' TEST
```

gives us the pfa of TEST,

```
3 +
```

offsets us past the address and the count, to the beginning of the string (the letter "S"), and

```
7 TYPE
```

types the string "SAMPLE."

That little exercise may not seem too useful. But let's go a step further.

Remember how we defined LABEL in our egg-sizing application, using `IF`... statements? We can rework our definition using `TABLE`. First let's make all the labels the same length and "string them together" within a single definition as a string array. (We can abbreviate the longest label to "XTRA LRG" so that we can make each label eight characters long, including trailing spaces.)

```
: "LABEL"
   ." REJECT SMALL MEDIUM LARGE XTRA LRGERROR " ;
```

Once we enter

```
' "LABEL" 3 +
```

to get the address of the start of the string, we can type any particular label by offsetting into the array. For example, if we want label 2, we simply add sixteen (2 x 8) to the starting address and type the eight characters of the name:

```
16 + 8 TYPE
```

Now let's redefine LABEL so that it takes a category-number from zero through five and uses it to index into the string array, like this:

```
: LABEL 8 * ['] "LABEL" 3 + + 8 TYPE SPACE ;
```

Recall that the word `[']` is just like `[` except that it may only be used inside a definition to compile the address of the next

word in the definition (in this case, "LABEL").[†] Later, when we execute LABEL, bracket-tick-bracket will push the pfa of "LABEL" onto the stack. The number three is added, then the string offset is added to compute the address of the particular label name that we want.

—This kind of string array is sometimes called a "superstring." As a naming convention, the name of the superstring usually has quotes around it.

Our new version of LABEL will run a little faster because it does not have to perform a series of comparison tests before it hits upon the number that matches the argument. Instead it uses the argument to compute the address of the appropriate string to be typed.

Notice, though, that if the argument to LABEL exceeds the range zero through five, you'll be typing garbage. If LABEL is only going to be used within EGGSIZE in the application, there's no problem. But if an "end user," meaning a person, is going to use it, you'd better "clip" the index, like this:

```
: LABEL 0 MAX 5 MIN LABEL ;
```

TYPE	(adr u --)	Transmits u characters, beginning at address, to the current output device.
------	-------------	---

[†]FORTH-79 Standard

See Appendix 3.

Outputting Strings from Disk

We mentioned before that the word `BLOCK` copies a given block into an available buffer and leaves the address of the buffer on the stack. Using this address as a starting-point, we can index into one of the buffer's 1,024 bytes and type any string we care to. For example, to print line 0 of block 214, we could say

```
CR 214 BLOCK 64 TYPE RETURN
( THIS IS BLOCK 214) ok
```

To print line eight, we could add 512 (8 x 64) to the address, like this:

```
CR 214 BLOCK 512 + 64 TYPE
```

Before we give a more interesting example, it's time to introduce two words that are closely associated with `TYPE`.

<pre>-TRAILING (adr u1 -- adr u2)</pre>	<pre>Eliminates trailing blanks from the string that starts at the address by reducing the count from u1 (original byte count) to u2 (shortened byte count).</pre>	<div style="border: 1px solid black; border-radius: 50%; padding: 2px; display: inline-block;">not-trailing</div>
<pre>>TYPE† (adr u --)</pre>	<pre>Same as TYPE except that the output string is moved to the pad prior to output. Used in multiprogrammed systems to output strings from disk blocks.</pre>	<div style="border: 1px solid black; border-radius: 50%; padding: 2px; display: inline-block;">bracket-type</div>



`-TRAILING` can be used immediately before the `TYPE` command to adjust the count so that trailing blanks will not be printed. For instance, inserting it into our first example above would give us

```
CR 214 BLOCK 64 -TRAILING TYPE RETURN
( THIS IS BLOCK 214) ok
```

†FORTH-79 Standard

`>TYPE` is not required.

The word `>TYPE` is only used on multiprogrammed systems to print strings from disk buffers. Instead of typing the string directly from the address given, it first moves the entire string into the pad, then types it from there. Because all users share the same buffers, the system cannot guarantee that by the time `TYPE` has finished typing, the buffer will still contain the same block. It can guarantee, however, that the buffer will contain the same block during the move to the pad.[†] Since each task has its own pad, `>TYPE` can safely type from there.

The following example uses `TYPE`, but you may substitute `>TYPE` if need be.

231 LIST

```

0 ( BUZZPHRASE GENERATOR -- VER. 1)      EMPTY
1
2 181 LOAD ( RANDOM NUMBERS)
3
4 : BUZZ 232 BLOCK + 10 CHOOSE 64 * + 20 -TRAILING TYPE ;
5 : 1ADJ 0 BUZZ ;
6 : 2ADJ 20 BUZZ ;
7 : NOUN 40 BUZZ ;
8 : PHRASE 1ADJ SPACE 2ADJ SPACE NOUN ;
9 : PARAGRAPH
10      CR ." BY USING " PHRASE ." COORDINATED WITH "
11      CR PHRASE ." IT IS POSSIBLE FOR EVEN THE MOST "
12      CR PHRASE ." TO FUNCTION AS "
13      CR PHRASE ." WITHIN THE CONSTRAINTS OF "
14      CR PHRASE ." . " ;
15 PARAGRAPH

```

(continued)

[†]For Experts

In a multiprogrammed system, a task only releases control of the CPU to the next task during I/O or upon explicit command, a command which is deliberately left out of the definition of the word which moves strings.

232 LIST

0	INTEGRATED	MANAGEMENT	CRITERIA
1	TOTAL	ORGANIZATION	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			
12			
13			
14			
15			

Upon loading the application block (in this case block 231), we get something like the following output, although some of the words will be different every time we execute PARAGRAPH.

BY USING INTEGRATED POLICY THROUGH-PUT COORDINATED WITH COMPATIBLE ORGANIZATION CAPABILITY IT IS POSSIBLE FOR EVEN THE MOST OPTIMAL THIRD GENERATION PROGRAMMING TO FUNCTION AS SYSTEMATIZED MONITORED CRITERIA WITHIN THE CONSTRAINTS OF RESPONSIVE POLICY HARDWARE.

As you can see, the definition of PARAGRAPH consists of a series of "." strings interspersed with the word PHRASE. If we execute PHRASE alone, we get

PHRASE SYSTEMATIZED MANAGEMENT MOBILITY ok

that is, one word chosen randomly from column 1 in block 232, one word from column 2, and one from column 3.

Looking at the definition of PHRASE, we see that it consists of three application words, 1ADJ, 2ADJ, and NOUN, each of which in turn consists of an offset and the application word BUZZ. The offset indicates which column we want to choose a particular word from; that is, the number of bytes in from the left margin of block 232 that the column begins. The definition of BUZZ breaks down as follows:

232 BLOCK

moves block 232 into an available buffer and returns the address of the buffer's beginning byte.

The word

+

adds the offset (0, 20, or 40) to offset us into the appropriate column in the block.

10 CHOOSE

returns a random number† between 0 and 10 to determine which line to take our word from.

64 * +

multiplies the random number by 64 (the length of one line) and adds this number to the buffer address, to offset into the appropriate line. The address on the stack is the address of the word we are going to type.

20 -TRAILING TYPE

adjusts the maximum count of 20 downwards so that the count excludes any trailing blanks after the character string and types the string.

†The random number generator is given in the following Handy Hint.

A Handy HintA Random Number Generator

This simple random number generator can be useful for games, although for more sophisticated applications such as simulations, better versions are available.

```
181 LIST
```

```
0 ( RANDOM NUMBER GENERATOR -- HIGH LEVEL )
1 VARIABLE RND   HERE RND !
2 : RANDOM   RND @ 31421 * 6927 + DUP RND ! ;
3 : CHOOSE   ( u1 --- u2 )
4           RANDOM U* SWAP DROP ;
5
6 ( where CHOOSE returns a random integer within the range
7   0 = or < u2 < u1. )
8
9
```

Here's how to use it:

To choose a random number between zero and ten (but exclusive of ten) simply enter


```
10 CHOOSE
```

and CHOOSE will leave the random number on the stack.

Internal String Operators

The commands for moving character strings or data arrays are very simple. Each requires three arguments: a source address, a destination address, and a count.

MOVE [†]	(adr1 adr2 u --)	Copies a region of memory u bytes long, cell-by-cell beginning at adr1, to memory beginning at adr2. The move begins with the contents of adr1 and proceeds toward high memory.	move
CMOVE	(adr1 adr2 u --)	Copies a region of memory u bytes long, byte-by-byte beginning at adr1, to memory beginning at adr2. The move begins with the contents of adr1 and proceeds toward high memory.	c-move
<CMOVE	(adr1 adr2 u --)	Copies a region of memory u bytes long, beginning at adr1, to memory beginning at adr2, but starts at the <u>end</u> of the string and proceeds toward low memory.	back-c-move



[†]FORTH-79 Standard

The Standard's `MOVE` expects a cell count. `<CMOVE` is not required.

Notice that these commands follow certain conventions we've seen before:

1. When the arguments include a source and a destination (as they do with `[COPY]`), the source precedes the destination.
2. When the arguments include an address and a count (as they do with `[---E]`), the address precedes the count.

And so with these three words the arguments are

(source destination count --)

To move the entire contents of a buffer into the pad, for example, we would write

210 BLOCK PAD 1024 CMOVE

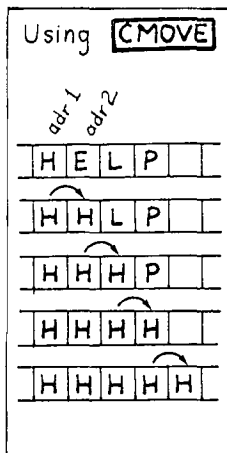
although on cell-address machines the move might be made faster if it were cell-by-cell, like this:

210 BLOCK PAD 1024 MOVE

The word `<CMOVE` lets you move a string to a region that is higher in memory but that overlaps the source region.†

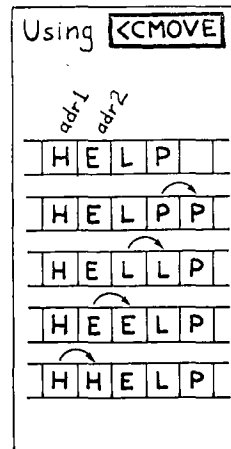
†For beginners

Let's say that you want to move a string one byte to the "right" in memory (e.g., when you use the editor command `I` to insert a character).



If you were to use `[CMOVE]`, the first letter of the string would get copied to the second byte, but that would "clobber" the second letter of the string. The final result would be a string composed of a single character.

Using `<CMOVE` in this situation keeps the string from clobbering itself during the move.



To blank an array, we can use the word `FILL`, which we introduced earlier. For example, to store blanks into 1024 bytes of the pad, we say

```
PAD 1024 32 FILL
```

Thirty-two is the ASCII representation of blank.[†]

Single-character Input

The word `KEY` awaits the entry of a single key from your terminal keyboard and leaves the character's ASCII equivalent on the stack in the low-order byte.

To execute it directly, you must follow it with a return, like this:

```
KEY RETURN
```

The cursor will advance a space, but the terminal will not print "ok"; it is waiting for your input. Press the letter "A," for example, and the screen will "echo" the letter "A," followed by the "ok." The ASCII value is now on the stack, so enter `]`:

```
~KEY Aok  
.RETURN 65 ok
```

This saves you from having to look in the table to determine a character's ASCII code.

You can also include `KEY` inside a definition. Execution of the definition will stop, when `KEY` is encountered, until an input character is received. For example, the following definition will list a given number of blocks in series, starting with the current block, and wait for you to press any key before it lists the next one:

```
: BLOCKS ( count -- )  
  SCR @ + SCR @ DO I LIST KEY DROP LOOP ;
```

[†]For polyFORTH Users

You may use the word `BLANK` instead, as in

```
PAD 1024 BLANK
```


A Handy Hint

Two Convenient Additions to the Editor

You might want to make the following two additions to your editor vocabulary. The use of these words is a matter of preference; they may or may not already be included with your system.

EDITOR DEFINITIONS

```
: K #I PAD 132 MOVE PAD #F 66 MOVE ;
: WIPE SCR @ BLOCK DUP 1024 32 FILL 0 SWAP ! UPDATE ;
FORTH DEFINITIONS
```

The word K will swap the contents of the find buffer with that of the insert buffer. Here's an example of its use:

```
^YOU HAVE THE RIGHT TO SILENT REMAIN. ok
DØSILENT RETURN
K
F AIN RETURN

YOU HAVE THE RIGHT TO REMAIN^ ok
I

YOU HAVE THE RIGHT TO REMAIN SILENT. ok
```

Use of **D** put "SILENT" in the find buffer, and K put it into the insert buffer so that you could insert it where it belongs.

Or if you've just inserted a string in the wrong place, you can put the string into the find buffer with K and then erase it from the line with a simple **E**.

The word WIPE blanks the current block and stores two nulls in the first two character positions. (On most systems, nulls in the block act just like the word **EXIT**, to immediately terminate interpretation of the block, should it be loaded.)

In this case we `DROP` the value left by `KEY` because we do not care what it is.

Or we might add a feature that allows us either to leave the loop at any time by pressing return or to continue by pressing any other key, such as space. In this case we will perform a conditional test on the value returned by `KEY`.

```
: BLOCKS ( count -- )
  SCR @ + SCR @ DO I LIST
  KEY 0= ( CR) IF LEAVE THEN LOOP ;
```

Note that in most FORTH systems, the carriage-return key is received as a null (zero).

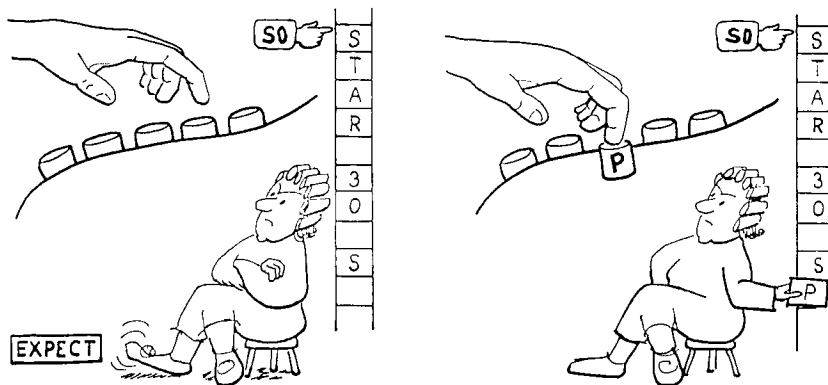
KEY	(-- c)	Returns the ASCII value of the next available character from the current input device.
-----	---------	--

String Input Commands, from the Bottom up

There are several words involved with string input. We'll start with the lowest-level of these and proceed to some higher-level words. Here are the words we'll cover in this section:

EXPECT	(adr u --)	Awaits u characters (or a carriage return) from the terminal keyboard and stores them, starting at the address.
WORD	(c -- adr)	Reads one word from the input stream, using the character (usually blank) as a delimiter. Moves the string to the address (HERE) with the count in the first byte, leaving the address on the stack.
TEXT	(c --)	Reads a string from the input stream, using the character as a delimiter, then sets the pad to blanks and moves the string to the pad.

The word `EXPECT` stops execution of the task and waits for input from your keyboard. It expects a given number of keystrokes or a carriage return, whichever comes first. The incoming text is stored beginning at the address given as an argument.



For example, the phrase

```
SO @ 80 EXPECT †
```

will await up to eighty characters and store them in the input message buffer.

This phrase is the one used in the definition of `QUIT` to get the input for `[IN1...]`.

In most systems, when you press return or when the limit is reached, `EXPECT` stores a null (zero) into the string to mark the end, then allows execution to continue.‡

†FORTH-79 Standard

This phrase is equivalent to the Standard word `[QU...]`.

‡For Experts

You can use `EXPECT` to accept data from a serial line, such as a measuring device. Since you supply the address and count, such data can be read directly into an array. In a single-user environment, you may read data into a buffer for storage on disk. In a multi-user environment, however, you must use `SO` and later move the data into the buffer, since another task may use "your" buffer.

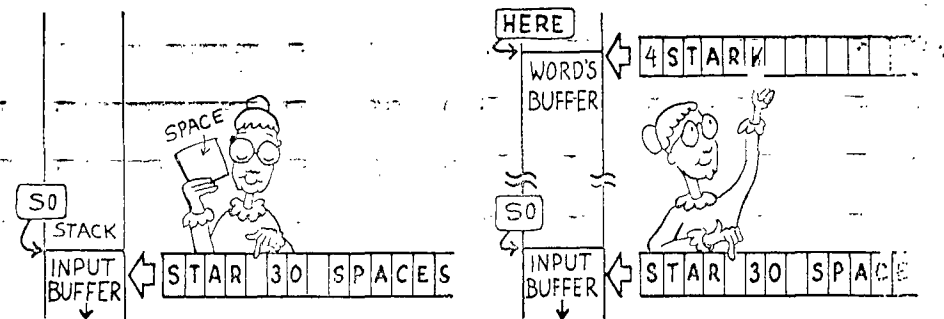
Let's move on to the next higher-level string-input operator. We've just explained that `QUIT` contains the phrase

```
... S0 @ 80 EXPECT INTERPRET ...
```

But how does the text interpreter scan the input message buffer and pick out each individual word there? With the phrase

```
32 WORD
```

The decimal number 32 is the ASCII representation for "space." `WORD` scans the input stream looking for the given delimiter, in this case space, and moves the sub-string into a different buffer of its own, with the count in the first byte of the buffer. Finally, it leaves the address of its buffer on the stack, so that `INTERPRET` (or anyone else) knows where to find it. `WORD`'s buffer usually begins at `H`, the dictionary pointer, so the address given is `HERE`.

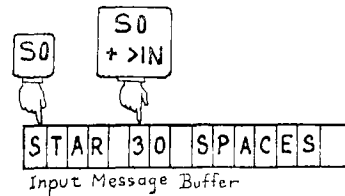


`WORD` looks for the given delimiter in the input message buffer,

and moves the sub-string to `WORD`'s buffer, with the count in the first byte.

When you are executing words directly from a terminal, `WORD` will scan the input buffer, starting at `S0`. As it goes along, it advances the input buffer pointer, called `>IN`, so that each time you execute `WORD`, you scan the next word in the input stream.

`>IN` is a "relative pointer"; that is, it does not contain the actual address but rather an offset that is to be added to the actual address, which in this case is `S0`. For example, after `WORD` has scanned the string "STAR," the value of `>IN` is five.



`WORD` ignores initial occurrences of the delimiter (until any other character is encountered). You could type

```
    ␣␣␣␣STAR
```

(that is, `STAR` preceded by several spaces) and get exactly the same string in `WORD`'s buffer as shown above.

When `WORD` moves the sub-string, it includes a blank at the end but does not include it in the count.

We'll get back to `WORD` later on in this chapter. For now, though, let's look at a word that uses `WORD` and that is more useful for handling string input.

`TEXT`,[†] like `WORD`, takes a delimiter and scans the input stream until it finds the string delimited by it. It then moves the string to the pad. What is especially nice about `TEXT` is that before it moves the string, it blanks the pad for at least sixty-four spaces. This makes it very convenient for use with `TEXT`. Here's a simple example:

```
CREATE MY-NAME 40 ALLOT
: I'M 32 TEXT PAD MY-NAME 40 CMOVE ;
```

In the first line we define an array called `MY-NAME`. In the second line we define a word called `I'M` which will allow us to enter

```
I'M EDWARD ok
```

[†]For Those Who Don't Seem to Have ...[T](#)

`TEXT` is not required by the FORTH-79 Standard. Its definition, however, is

```
: TEXT PAD 72 32 FILL WORD COUNT PAD SWAP <CMOVE ;
```

If you have a polyFORTH system, the electives block normally does not load the block (usually 34) that contains `TEXT`. In this case you must add "34 LOAD" to your electives block and reload it.

The definition of I'M breaks down as follows: the phrase

```
32 TEXT
```

scans the remainder of the input stream looking for a space or for the end of the line, whichever comes first. (The delimiter that we give as an argument to `TEXT` is actually used by `WORD`, which is included in the definition of `TEXT`.) `I` then moves the phrase to a nice clean "pad."

The phrase

```
PAD MY-NAME 40 CMOVE
```

moves forty bytes from the pad into the array called MY-NAME, where it will safely stay for as long as we need it.

We could now define GREET as follows:

```
: GREET ." HELLO, " MY-NAME 40 -TRAILING TYPE
      ." , I SPEAK FORTH. " ;
```

so that by executing GREET, we get

```
GREET HELLO, EDWARD, I SPEAK FORTH. ok-
```

Unfortunately, our definition of I'M is looking for a space as its delimiter. This means that a person named Mary Kay will not get her full name into MY-NAME.

To get the complete input stream, we don't want to "see" any delimiter at all, except the end of the line. Instead of "32 TEXT," we should use the phrase

```
1 TEXT
```

ASCII 1 is a control character that can't be sent from the keyboard and therefore won't ever appear in the input buffer. Thus "1 TEXT" is a convention used to read the entire input buffer, up to the carriage return. By redefining I'M in this way, Mary Kay can get her name into MY-NAME, space and all.

By using other delimiters, such as commas, we can "expect" a series of strings and store each of them into a different array for different purposes. Consider this example, in which the word VITALS uses commas as delimiters to separate three input fields:

233 LIST

```

0 ( FORM LOVE LETTER)          EMPTY
1 VARIABLE NAME 12 ALLOT      VARIABLE EYES 10 ALLOT
2 VARIABLE ME 12 ALLOT
3 : VITALS 44 TEXT ( , ) PAD NAME 14 MOVE
4           44 TEXT          PAD EYES 12 MOVE
5           1 TEXT          PAD ME 14 MOVE ;
6
7 : LETTER PAGE
8   ." DEAR " NAME 14 -TRAILING TYPE ." ,"
9   CR ." I GO TO HEAVEN WHENEVER I SEE YOUR DEEP "
10          EYES 12 -TRAILING TYPE ." EYES. CAN "
11   CR ." YOU GO TO THE MOVIES FRIDAY? "
12          CR 30 SPACES ." LOVE,"
13          CR 30 SPACES ME 14 -TRAILING TYPE
14   CR ." P.S. WEAR SOMETHING " EYES 12 -TRAILING TYPE
15   ." TO SHOW OFF THOSE EYES! " ;

```

which allows you to enter

```
VITALS ALICE,BLUE,FRED ok
```

then enter

```
LETTER
```

It works every time.

So far all of our input has been "FORTH style"; that is, numbers precede commands (so that a command will find its number on the stack) and strings follow commands (so that a command will find its string in the input stream). This style makes use of one of FORTH's unique features: it awaits your commands; it does not prompt you.

But if you want to, you may put EXPECT inside a definition so that it will request input from you under control of the definition. For example, we could combine the two words I'M and GREET into a single word which "prompts" users to enter their names. For example,

```
GREET
WHAT'S YOUR NAME?
```

at which point execution stops so the user can enter a name:

```
GREET
WHAT'S YOUR NAME? TRAVIS MC GEE
HELLO, TRAVIS : GEE, I SPEAK FORTH. ok
```

We could do this as follows:

```
: GREET CR ." WHAT'S YOUR NAME?" SO @ 40 EXPECT
      0 >IN ! 1 TEXT CR ." HELLO, "
      PAD 40 -TRAILING TYPE ." , I SPEAK FORTH. " ;
```

We've explained all the phrases in the above definition except this one:


```
----- 0 >IN !
```

Remember that `TEXT`, because it uses `WORD`, always uses `>IN` as its reference point. But when the user enters the word `GREET` to execute this definition, the string "GREET" will be stored in the input message buffer and `>IN` will be pointing beyond "GREET". `EXPECT` does not use `>IN` as its reference, so it will store the user's name beginning at `SO`, on top of `GREET`. If you were to execute `TEXT` now, it would miss the first five letters of the user's name. It's necessary to reset `>IN` to zero so that `TEXT` will look where `EXPECT` has put the name.

Number Input Conversions

When you type a number at your terminal, FORTH automatically converts this character string into a binary value and pushes it onto the stack. FORTH also provides two commands which let you convert a character string that begins at any memory location into a binary value.†

>BINARY or CONVERT	(dl adr1 -- d2 adr2)	Converts the text beginning at adr1+1 to a binary value with regard to BASE. The new value is accumulated into dl, being left as d2; adr2 is the address of the first non-convertible character.	to- binary
NUMBER	(adr -- n or d)	Converts the text beginning at adr+1, with regard to BASE, to a binary value that is single-length if no valid punctuation occurs and double-length if valid punctuation does occur. The string may contain a preceding negative sign; adr may contain a count, which will be ignored.	number



`NUMBER` exists on most systems and is usually the simpler to use. Here's an example that uses `NUMBER`:

```
: PLUS 32 WORD NUMBER + ." = " . ;
```

PLUS allows us to prove to any skeptic that FORTH could use infix notation if it wanted to. We can enter

† FORTH-79 Standard

The Standard specifies the name `CONV` instead of `>BINARY`. In FORTH systems which use three-character uniqueness, however, this choice conflicts with the name `CONV`; hence the name `>BINARY` is used instead. `NUMBER` is not required by the Standard.

2 PLUS 13 RETURN = 15 ok

When PLUS is executed, the "2" will be on the stack in binary form, while the "3" will still be in the input stream as a string. The phrase

32 WORD

reads the string; NUMBER converts it to binary and puts the value on the stack; + adds the two values; and . prints the sum.

NUMBER expects on the stack the address of the string that is to be converted, with the count in the first byte and one trailing blank, so it's most appropriate for use after WC. NUMBER does not actually use the count, however; it only adds one byte to the address before beginning the conversion. Thus you can use NUMBER on a string that does not contain the count in the first byte, simply by subtracting one byte from the starting address of the string.

>BINARY is a more primitive definition, being used in the definition of NUMBER. You can use >BINARY to create your own specialized number input conversion routines. Since >BINARY returns the address of the first non-convertible character, you can make decisions based on whether the character is a hyphen, dot, or whatever. You can also make decisions based on the location of the non-convertible character within the number. For instance, you can write a routine that lets you enter a number with a decimal point in it and then scales it accordingly.

To give a good example of the use of >BINARY, Figure 10-1 shows a definition of NUMBER. This version reads any of the characters

: , - . /

as valid punctuation characters which cause the value to be returned on the stack as a double-length integer. If none of these characters appear in the string, the value is returned as single-length.† This definition uses the word WITHIN as we defined it in the problems for Chap. 4.

Here we use the variable PUNCT to contain a flag that indicates whether punctuation was encountered. We suggest that you use an available user variable instead.

†For polyFORTH Users

Your version of NUMBER behaves similarly and in addition leaves in the user variable PTR the number of characters that were converted since the last punctuation was encountered.

FIGURE 10-1. A DEFINITION OF NUMB...

VARIABLE PUNCT	Creates a flag that will contain true if the number contains valid punctuation.
: NUMBER (adr -- n or d)	
0 PUNCT !	Initializes flag: no punctuation has occurred.
DUP 1+ C@	Gets the first digit.
45 (-) =	Is it a minus sign?
DUP >R	Saves the flag on the return stack.
+	If the first character is "-", adds 1 (the flag itself) to the address, setting it to point to the first digit.
0 0 ROT	Provides a double-length zero as an accumulator.
BEGIN >BINARY	Begins conversion; converts until an invalid digit.
DUP C@	Fetches the invalid digit.
32 - WHILE	While it is not a blank, checks if it is valid punctuation; that is,
DUP C@ DUP 58 =	a colon, or
SWAP 44 48 WITHIN +	a comma, hyphen, period, or slash.
DUP PUNCT !	Sets PUNCT to indicate whether valid punctuation has occurred.
NOT ABORT" ? "	Otherwise issues an error message.
REPEAT	Exits here if a blank is detected; otherwise repeats conversion.
DROP	Discards the address on the stack.
R> IF DNEGATE THEN	If the flag on the return stack is true, negates d.
PUNCT @ NOT IF DROP THEN ;	If there was no punctuation, returns a single-length value by dropping the high-order cell.

A Closer Look at `WORD`

So far we have only talked about using `WORD` to scan the input message buffer (which holds the characters that are `EXPECT`ed from the terminal). But if we recall that the phrase

```
32 WORD
```

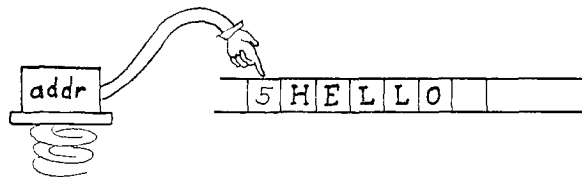
is used by the text interpreter, we realize that `WORD` actually scans the input stream, which is either the input message buffer or a block buffer that is being `LOAD`ed.

To achieve this flexibility, `WORD` uses another pointer in addition to `>IN`, called `BLK` (pronounced b-l-k). `BLK` acts both as a flag and as a pointer. If `BLK` contains zero, then `WORD` scans the input message buffer (that is, offset by `>IN`). But if `BLK` contains a non-zero number, then `WORD` is referring to a block buffer and the number in `BLK` is the number of the block. Here are two examples:

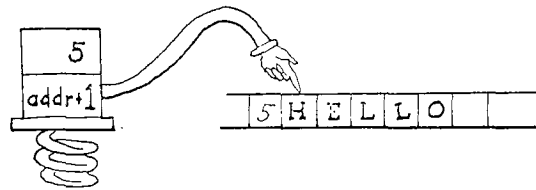
contents of BLK	address currently used by <code>WORD</code> :
0	50 @ >IN @ + (>IN bytes into the message buffer)
200	200 BLOCK >IN @ + (>IN bytes into the block buffer)

Every time a word is interpreted during a `LOAD` operation, `WORD` makes sure that the appropriate block is still in a buffer.

A useful word to use in conjunction with `WORD` is `COUNT`. Recall that `WORD` leaves the length of the word in the first byte of `WORD`'s buffer and also leaves the address of this byte on the stack.



The word `COUNT` puts the count on the stack and increments the address, like this:



leaving the stack with a string address and a count as appropriate arguments for `TYPE`, `CMOVE`, etc.

`COUNT` is used in the definition of `TEXT` which we gave in a footnote earlier.

<code>COUNT</code>	<code>(adr -- adr+1 u)</code>	Converts a character string, whose length is contained in its first byte, into the form appropriate for <code>TYPE</code> , by leaving the address of the first character and the length on the stack.
--------------------	-------------------------------	--

We will further illustrate the use of `[W[.]]` in one of the examples in Chap. 12.

String Comparisons

Here is a FORTH word that you can use to compare character strings:

<code>-TEXT</code>	<code>(adr1 u adr2 -- f)</code>	Compares two strings that start at <code>adr1</code> and <code>adr2</code> , each of length <code>u</code> . Returns false if they match; true if no match (positive if binary string 1 > 2, negative if 1 < 2).
--------------------	---------------------------------	--

not-text



`[-TEXT]` can be used to test either whether two character strings are equal or whether one is alphabetically greater or lesser than the other.†‡ Chap. 12 includes an example of using `[-TEXT]` to determine whether strings match exactly.

Since for speed `[-TEXT]` compares cell-by-cell, you must take care on cell-address machines to give `[-TEXT]` even cell addresses only. For example, if you want to compare a string that is being entered as input with a string that is in an array, bring the input string to the pad (using `[TEXT]` rather than `[WORD]`) because `[PAD]` is an even address. Similarly, if you want to test a string that is in a block buffer, you must either guarantee that the string's address is even or, if you cannot know for sure, move the string to an even address (using `[CMOVE]`) before making the test.

By the way, the hyphen in `[-TEXT]` is as close as ASCII comes to "-", the logical symbol meaning "not." This is why we conventionally use this prefix for words which return a "negative true" flag. (Negative true means that a zero represents true and a non-zero represents false.) We pronounce such words not-text, etc.

† For Users of Intel, DEC, and Zilog Processors

To make the "alphabetical" test, you must first reverse the order of bytes.

‡ FORTH-79 Standard

`[-TEXT]` is not included in the Standard. If your system does not have `[-TEXT]`, you can load the high-level definition below. Of course, `[-TEXT]` is written in assembler code on all polyFORTH systems, for speed.

```

: -TEXT  2DUP + SWAP DO  DROP  2+
        DUP 2- @ I @ - DUP IF DUP ABS / LEAVE THEN.
        2 +LOOP SWAP  DROP ;

```

Here's a list of the FORTH words covered in this chapter.

UPDATE	(--)	Marks the most recently referenced block as modified. The block will later be automatically transferred to mass storage if its buffer is needed to store a different block or if FLUSH is executed.
EMPTY-BUFFERS	(--)	Marks all block buffers as empty without necessarily affecting their actual contents. Updated blocks are not written to mass storage.
BLOCK	(u -- adr)	Leaves the address of the first byte in block u. If the block is not already in memory, it is transferred from mass storage into whichever memory buffer has been least recently accessed. If the block occupying that buffer has been updated (i. e., modified), it is rewritten onto mass storage before block u is read into the buffer.
BUFFER	(u -- adr)	Obtains the next block buffer, assigning it to block u. The block is not read from mass storage.
TYPE	(adr u --)	Transmits u characters, beginning at address, to the current output device.
-TRAILING	(adr u1 -- adr u2)	Eliminates trailing blanks from the string that starts at the address by reducing the count from u1 (original byte count) to u2 (shortened byte count).

MOVE	(adr1 adr2 u --)	Copies a region of memory u bytes long, cell-by-cell beginning at adr1, to memory beginning at adr2. The move begins with the contents of adr1 and proceeds toward high memory.
CMOVE	(adr1 adr2 u --)	Copies a region of memory u bytes long, byte-by-byte beginning at adr1, to memory beginning at adr2. The move begins with the contents of adr1 and proceeds toward high memory.
KEY	(-- c)	Returns the ASCII value of the next available character from the current input device.
EXPECT	(adr u --)	Awaits u characters (or a carriage return) from the terminal keyboard and stores them, starting at the address.
WORD	(c -- adr)	Reads one word from the input stream, using the character (usually blank) as a delimiter. Moves the string to the address (HERE) with the count in the first byte, leaving the address on the stack.
TEXT	(c --)	Reads a string from the input stream, using the character as a delimiter, then sets the pad to blanks and moves the string to the pad.
>BINARY or CONVERT	(dl adr1 -- d2 adr2)	Converts the text beginning at adr1+1 to a binary value with regard to BASE. The new value is accumulated into dl, being left as d2; adr2 is the address of the first non-convertible character.

NUMBER	(adr -- n or d)	Converts the text beginning at adr+1, with regard to BASE, to a binary value that is single-length if no valid punctuation occurs, and double-length if valid punctuation does occur. The string may contain a preceding negative sign; adr may contain a count, which will be ignored.
COUNT	(adr -- adr+1 u)	Converts a character string, whose length is contained in its first byte, into the form appropriate for TYPE, by leaving the address of the first character and the length on the stack.
Additional Words Available in Some Systems		
>TYPE	(adr u --)	Same as TYPE except that the output string is moved to the pad prior to output. Used in multiprogrammed systems to output strings from disk blocks.
<CMOVE	(adr1 adr2 u --)	Copies a region of memory u bytes long, beginning at adr1, to memory beginning at adr2, but starts at the <u>end</u> of the string and proceeds toward low memory.
-TEXT	(adr1 u adr2 -- f)	Compares two strings that start at adr1 and adr2, each of length u. Returns false if they match; true if no match (positive if binary string 1 > 2, negative if 1 < 2).
BLANK	(adr n --)	Stores ASCII blanks into n bytes of memory, beginning at adr.

Review of Terms

<u>Buffer status cell</u>	in the FORTH operating system, a cell in resident memory associated with each block buffer (usually directly preceding it in memory) which contains the number of the block currently stored in the buffer and a flag (the sign bit) which indicates whether the buffer has been updated.
<u>Relative pointer</u>	a variable which specifies a location in relation to the beginning of an array or string --not the absolute address.
<u>Superstring</u>	in FORTH, a character array which contains a number of strings. Any one string may be accessed by indexing into the array.
<u>Virtual memory</u>	the treatment of mass storage (such as the disk) as though it were resident memory; also the mechanisms of the operating system which make this treatment possible.

Problems -- Chapter 10

1. Enter some famous quotations into an available block, say 228. Now define a word called CHANGE which takes two ASCII values and changes all occurrences within block 228 of the first character into the second character. For example,

```
65 69 CHANGE
```

will change all the "A"s into "E"s.

2. Define a word called FORTUNE which will print a prediction at your terminal, such as "You will receive good news in the mail." The prediction should be chosen at random from a list of sixteen or fewer predictions. Each prediction is sixty-four characters, or less, long.
3. According to Oriental legend, Buddha endows all persons born in each year with special, helpful characteristics represented by one of twelve animals. A different animal reigns over each year, and every twelve years the cycle repeats itself. For instance, persons born in 1900 are said to be born in the "Year of the Rat." The art of fortune-telling based on these influences of the natal year is called "Juneeshee."

Here is the order of the cycle:

```
Rat Ox Tiger Rabbit Dragon Snake
Horse Ram Monkey Cock Dog Boar
```

Write a word called .ANIMAL that types the name of the animal corresponding to its position in the cycle as listed here; e.g.,

```
0 .ANIMAL RAT ok
```

Now write a word called (JUNESHEE) which takes as an argument a year of birth and prints the name of the associated animal. (1900 is the year of the Rat, 1901 is the Ox, etc.)

Finally, write a word called JUNESHEE which prompts the user for his/her year of birth and prints the name of the person's Juneeshee animal. Define it so the user won't have to press "return" after entering the year.

4. Rewrite the definition of LETTER that appears in this chapter so that it uses names and personal descriptions that have been edited into a block, rather than entered into character arrays. In this way, you can keep a file on many "prospects" and produce a letter for any one person with the

appropriate descriptions, just by supplying an argument to LETTER, as in

```
1 LETTER
```

Now define LETTERS so that it prints one letter for each person in your file.

5. In this exercise you will create and use a virtual array, that is, an array which resides on the disk but which is referenced like a memory-resident array (with `@` and `!`).

First select an unused block in your range of assigned blocks. There can be no text on this block; binary data will be stored in it. Put this block number in a variable. Then define an access word which accepts a cell subscript from the stack, then computes the block number corresponding to this subscript, calls `BLOCK` and returns the memory address of the subscripted cell. This access word should also call `UPDATE`. Test your work so far.

Next use the first cell as a count of how many data items are stored in the array. Define a word `PUT` which will store a value into the next available cell of the array. Define a display routine which will print the stored elements in the array.

Now use this virtual array facility to define a word `ENTER` which will accept pairs of numbers and store them in the array.

Finally, define `TABLE` to print the data entered above, eight numbers per line.