7 A NUMBER OF KINDS OF NUMBERS

So far we've only talked about signed single-length numbers. In this chapter we'll introduce unsigned numbers and double-length numbers, as well as a whole passel of new operators to go along with them.

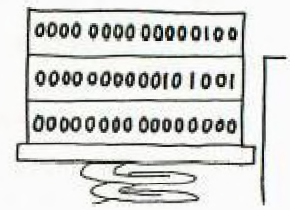
The chapter is divided into two sections:

For beginners--this section explains how a computer looks at numbers and exactly what is meant by the terms signed or unsigned and by single-length or double-length.

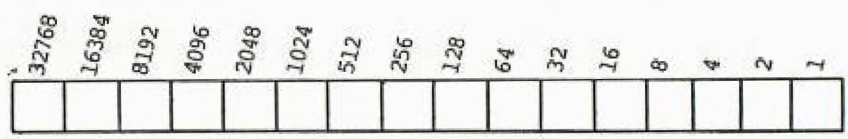
For everyone--this section continues our discussion of FORTH for beginners and experts alike, and explains how FORTH handles signed and unsigned, single- and double-length numbers.

SECTION I — FOR BEGINNERS

Signed vs. Unsigned Numbers

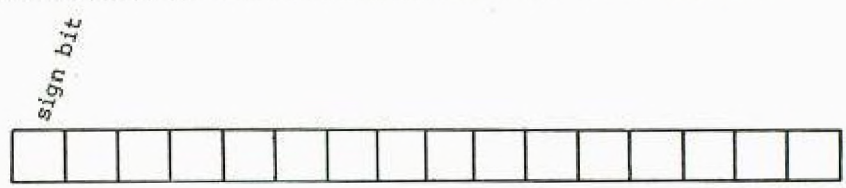


All digital computers store numbers in binary form.† In FORTH, the stack is sixteen bits wide (a "bit" is a "binary digit"). Below is a view of sixteen bits, showing the value of each bit:



If every bit were to contain a 1, the total would be 65535. Thus in 16 bits we can express any value between 0 and 65535. Because this kind of number does not let us express negative values, we call it an "unsigned number." We have been indicating unsigned numbers with the letter "u" in our tables and stack notations.

But what about negative numbers? In order to be able to express a positive or negative number, we need to sacrifice one bit that will essentially indicate sign. This bit is the one at the far left, the "high-order bit." In 15 bits we can express a number as high as 32767. When the sign bit contains 1, then we can go an equal distance back into the negative numbers. Thus within 16 bits we can represent any number from -32768 to +32767. This should look familiar to you as the range of a single-length number, which we have been indicating with the letter "n."



---

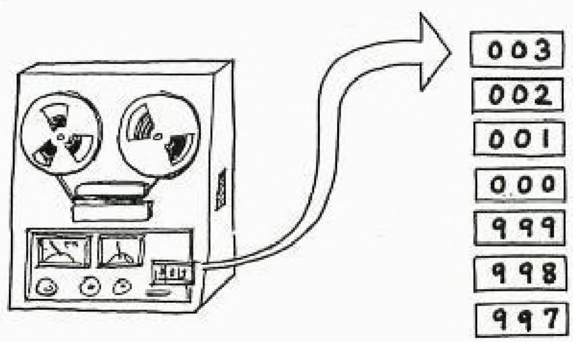
†For Beginner Beginners

If you are unfamiliar with binary notation, ask someone you know who likes math, or find a book on computers for beginners.

Before we leave you with any misconceptions, we'd better clarify the way negative numbers are represented. You might think that it's a simple matter of setting the sign bit to indicate whether a number is positive or negative, but it doesn't work that way.

To explain how negative numbers are represented, let's return to decimal notation and examine a counter such as that found on many tape recorders.

Let's say the counter has three digits. As you wind the tape forward, the counter-wheels turn and the number increases. Starting once again with the counter at 0, now imagine you're winding the tape backwards. The first number you see is 999, which, in a sense, is the same as -1. The next number will be 998, which is the same as -2, and so on.



The representation of signed numbers in a computer is similar.

Starting with the number

    0000000000000000

and going backwards one number, we get

    1111111111111111   (sixteen ones)

which stands for 65535 in unsigned notation as well as for -1 in signed notation. The number

    1111111111111110

which stands for 65534 in unsigned notation, represents -2 in signed notation.

Here's a chart that shows how a binary number on the stack can be used either as an unsigned number or as a signed number:

as an unsigned number		as a signed number
65535	1111111111111111	
...	...	
32768	1000000000000000	
32767	0111111111111111	32767
...	...	...
0	0000000000000000	0
	1111111111111111	-1
	...	...
	1000000000000000	-32768

This bizarre-seeming method for representing negative values makes it possible for the computer to use the same procedures for subtraction as for addition.

To show how this works, let's take a very simple problem:

```
 2
-1
```

Subtracting one from two is the same as adding two plus negative one. In single-length binary notation, the two looks like this:

    0000000000000010

while negative-one looks like this:

    1111111111111111

The computer adds them up the same way we would on paper; that is when the total of any column exceeds one, it carries a one into the next column. The result looks like this:

```
  0000000000000010
+ 1111111111111111
 10000000000000001
```

As you can see, the computer had to carry a one into every column all the way across, and ended up with a one in the seventeenth place. But since the stack is only sixteen bits wide, the result is simply

    0000000000000001

which is the correct answer, one.

We needn't explain how the computer converts a positive number to negative, but we will tell you that the process is called "two's complementing."

Arithmetic Shift

While we're on the subject of how a computer performs certain mathematical operations, we'll explain what is meant by the mysterious phrases back in Chap. 5: "arithmetic left shift" and "arithmetic right shift."

A FORTH Instant Replay:

2*	(n -- n*2)	Multiplies by two (arithmetic left shift).
2/	(n -- n/2)	Divides by two (arithmetic right shift).

To illustrate, let's pick a number, say six, and write it in binary form:

    0000000000000110

(4 + 2). Now let's shift every digit one place to the left, and put a zero in the vacant place in the one's column.

    0000000000001100

This is the binary representation of twelve (8 + 4), which is exactly double the original number. This works in all cases, and it also works in reverse. If you shift every digit one place to the <u>right</u> and fill the vacant digit with a zero, the result will always be <u>half</u> of the original value.

In arithmetic shift, the sign bit does not get shifted. This means that a positive number will stay positive and a negative number will stay negative when you divide or multiply it by two. (When the high-order bit shifts with all the other bits, the term is "logical shift.")

The important thing for you to know is that a computer can shift digits much more quickly than it can go through all the folderol of normal division or multiplication. When speed is critical, it's much better to say

```
2*
```

than

```
2 *
```

and it may even be better to say

```
2* 2* 2*
```

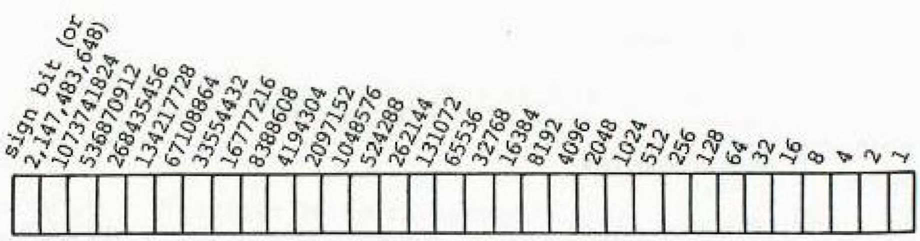
than

```
8 *
```

depending on your particular model of computer, but this topic is getting too technical for right now.

An Introduction to Double-length Numbers

A double-length number is just what you probably expected it would be: a number that is represented in thirty-two bits instead of sixteen. Signed double-length numbers have a range of $\pm$2,147,483,647 (a range of over four billion).



In FORTH, a double-length number takes the place of two single-length numbers on the stack. Operators like $\boxed{\text{2SWAP}}$ and $\boxed{\text{2DUP}}$ are useful either for double-length numbers or for pairs of single-length numbers.

One more thing we should explain: to the non-FORTH-speaking computer world, the term "word" means a 16-bit value, or two bytes. But in FORTH, "word" means a defined command. So in order to avoid confusion, FORTH programmers refer to a 16-bit value as a "cell." A double-length number requires two cells.

Other Number Bases

As you get more involved in programming, you'll need to employ other number bases besides decimal and binary, particularly hexadecimal (base 16) and octal (base 8). Since we'll be talking about these two number bases later on in this chapter, we think you might like an introduction now.

Computer people began using hexadecimal and octal numbers for one main reason: computers think in binary and human beings have a hard time reading long binary numbers. For people, it's much easier to convert binary to hexadecimal than binary to decimal, because sixteen is an even power of two, while ten is not. The same is true with octal. So programmers usually use hex or octal to express the binary numbers that the computer uses for things like addresses and machine codes. Hexadecimal (or simply "hex") looks strange at first since it uses the letters A through F.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Let's take a single-length binary number:

<u>0111101110100001</u>

To convert this number to hexadecimal, we first subdivide it into four units of four bits each:

| 0111 | 1011 | 1010 | 0001 |

then convert each 4-bit unit to its hex equivalent:

| 7 | B | A | 1 |

or simply 7BA1.

Octal numbers use only the numerals 0 through 7. Because nowadays most computers use hexadecimal representation, we'll skip an octal conversion example

We'll have more on conversions in the section titled "Number Conversions" later in this chapter.

The ASCII Character Set

If the computer uses binary notation to store numbers, how does it store characters and other symbols? Binary, again, but in a special code that was adopted as an industry standard many years ago. The code is called the American Standard Code for Information Interchange code, usually abbreviated ASCII.

Table 7-1 shows each character in the system and its numerical equivalent, both in hexadecimal and in decimal form.

The characters in the first column (ASCII codes 0-1F hex) are called "control characters" because they indicate that the terminal or computer is supposed to do something like ring its bell, backspace, start a new line, etc. The remaining characters are called "printing characters" because they produce visible characters including letters, the numerals zero through nine, all available symbols and even the blank space (hex 20). The only exception is DEL (hex 7F) which is a signal to the computer to ignore the last character sent.

In Chap. 1 we introduced the word EMIT. EMIT takes an ASCII code on the stack and sends it to the terminal so that the terminal will print it as a character. For example,

```
65 EMIT A ok
66 EMIT B ok
```

etc. (We're using the decimal, rather than the hex, equivalent because that's what your computer is most likely expecting right now.)†

Why not test EMIT on every printing character, "automatically"?

```
: PRINTABLES   127 32 DO I EMIT SPACE LOOP ;
```

---

†For Experts

Why are you snooping on the beginner's section?

TABLE 7-1 -- ASCII CHARACTERS & EQUIVALENTS

Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
NUL	00	0	SP	20	32	@	40	64	`	60	96
SOH	01	1	!	21	33	A	41	65	a	61	97
STX	02	2	"	22	34	B	42	66	b	62	98
ETX	03	3	#	23	35	C	43	67	c	63	99
EOT	04	4	$	24	36	D	44	68	d	64	100
ENQ	05	5	%	25	37	E	45	69	e	65	101
ACK	06	6	&	26	38	F	46	70	f	66	102
BEL	07	7	'	27	39	G	47	71	g	67	103
BS	08	8	(	28	40	H	48	72	h	68	104
HT	09	9	)	29	41	I	49	73	i	69	105
LF	0A	10	*	2A	42	J	4A	74	j	6A	106
VT	0B	11	+	2B	43	K	4B	75	k	6B	107
FF	0C	12	,	2C	44	L	4C	76	l	6C	108
CR	0D	13	-	2D	45	M	4D	77	m	6D	109
SM	0E	14	.	2E	46	N	4E	78	n	6E	110
SI	0F	15	/	2F	47	O	4F	79	o	6F	111
DLE	10	16	0	30	48	P	50	80	p	70	112
DC1	11	17	1	31	49	Q	51	81	q	71	113
DC2	12	18	2	32	50	R	52	82	r	72	114
DC3	13	19	3	33	51	S	53	83	s	73	115
DC4	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
FS	1C	28	<	3C	60	\	5C	92	|	7C	124
GS	1D	29	=	3D	61	]	5D	93	}	7D	125
RS	1E	30	>	3E	62	^	5E	94	~	7E	126
US	1F	31	?	3F	63	_	5F	95	DEL (RB)	7F	127

The "Char" columns list the ASCII characters (some of which are control characters); the "Hex" columns give the hexadecimal equivalents; and the "Dec" columns present the decimal equivalents.

PRINTABLES will emit every printable character in the ASCII set; that is, the characters from decimal 32 to decimal 126. (We're using the ASCII codes as our DO loop index.)

```
PRINTABLES  ! " # $ % & ' ( ) * + ... ok
```

Beginners may be interested in some of the control characters as well. For instance, try this:


```
7 EMIT ok
```

You should have heard some sort of beep, which is the video terminal's version of the mechanical printer's "typewriter bell."

Other control characters that are good to know include the following:

name	operation	decimal equivalent
BS	backspace	8
LF	line feed	10
CR	carriage return	13

Experiment with these control characters, and see what they do.

ASCII is designed so that each character can be represented by one byte. The tables in this book use the letter "c" to indicate a byte value that is being used as a coded ASCII character.

Bit Logic

The words AND and OR (which we introduced in Chap. 4) use "bit logic"; that is, each bit is treated independently, and there are no "carries" from one bit-place to the next. For example, let's see what happens when we AND these two binary numbers:

```
0000000011111111
0110010110100010  AND
----------------
0000000010100010
```

For any result-bit to be "1," the respective bits in both arguments must be "1." Notice in this example that the argument on top contains all zeroes in the high-order byte and all ones in the low-order byte. The effect on the second argument in this example is that the low-order eight bits are kept but the high-order eight bits are all set to zero. Here the first argument is being used as a "mask," to mask out the high-order byte of the second argument.

The word $\boxed{\text{OR}}$ also uses bit logic. For example,

```
 1000100100001001
 0000001111001000   [OR]
 ----------------
 1000101111001001
```

a "1" in either argument produces a "1" in the result. Again, each column is treated separately, with no carries.

By clever use of masks, we could even use a 16-bit value to hold sixteen separate flags. For example, we could find out whether this bit

```
 1011101010011100
            ▲
```

is "1" or "0" by masking out all other flags, like this:

```
 1011101010011100
 0000000000010000   [AND]
 ----------------
 0000000000010000
```

Since the bit was "1," the result is "true." Had it been "0," the result would have been "0" or "false."

We could set the flag to "0" without affecting the other flags by using this technique:

```
 1011101010011100
 1111111111101111   [AND]
 ----------------
 1011101010001100
            ▲
```

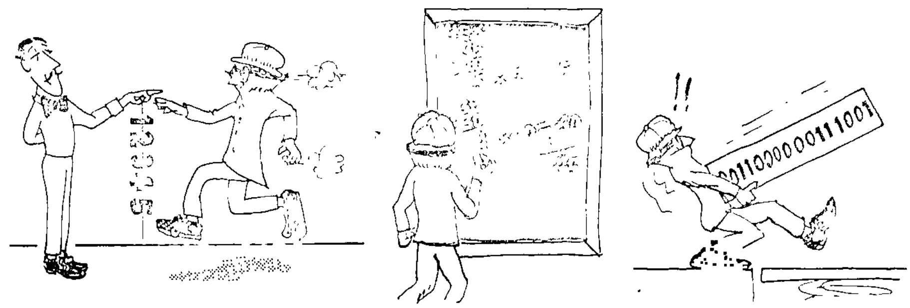
We used a mask that contains all "1"s except for the bit we wanted to set to "0." We can set the same flag back to "1" by using this technique:

```
 1011101010001100
 0000000000010000   [OR]
 ----------------
 1011101010011100
            ▲
```

SECTION II -- FOR EVERYBODY

Signed and Unsigned Numbers

Back in Chap. 1 we introduced the word NUMBER.



If the word INTERPRET can't find an incoming string in the dictionary, it hands it over to the word NUMBER. NUMBER then attempts to convert the string into a number expressed in binary form. If NUMBER succeeds, it pushes the binary equivalent onto the stack.

NUMBER does not do any range-checking.† Because of this, NUMBER can convert either signed or unsigned numbers.

For instance, if you enter any number between 32768 and 65535, NUMBER will convert it as an unsigned number. Any value between -32768 and -1 will be stored as a two's-complement integer.

This is an important point: the stack can be used to hold either signed or unsigned integers. Whether a binary value is interpreted as signed or unsigned depends on the operators that you apply to it. You decide which form is better for a given situation, then stick to your choice.

---

†For Beginners

This means that NUMBER does not check whether the number you've entered as a single-length number exceeds the proper range. If you enter a giant number, NUMBER converts it but only saves the least significant sixteen digits.

We've introduced the word [.], which prints a value on the stack as a signed number:

```
65535 . -1 ok
```

The word [U.] prints the same binary representation as an unsigned number:

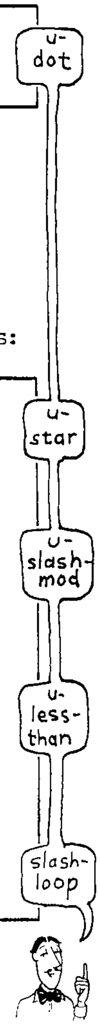
```
65535 U. 65535 ok
```

U.	(u --)	Prints the unsigned single-length number, followed by one space.

In this book the letter "n" signifies signed single-length numbers, while the letter "u" signifies unsigned single-length numbers. (We've already introduced [U.R], which prints an unsigned number right-justified within a given column width.)

Here is a table of additional words that use unsigned numbers:

U*	(u1 u2 -- ud)	Multiplies two 16-bit numbers. Returns a 32-bit result. All values are unsigned.
U/MOD	(ud u1 -- u2 u3)	Divides a 32-bit by a 16-bit number. Returns a 16-bit quotient and remainder. All values are unsigned.
U<	(u1 u2 -- f)	Leaves true if u1 < u2, where both are treated as 16-bit unsigned integers.
DO ... /LOOP†	DO: (u-limit u-index --) /LOOP: (u --)	Like DO ... +LOOP except uses an unsigned limit, index, and increment.



---

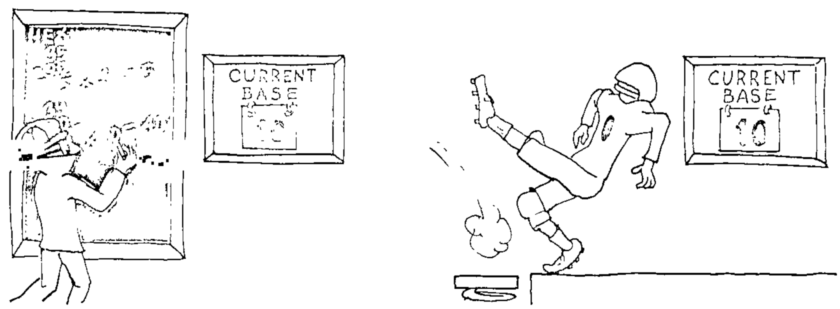
†FORTH-79 Standard

[/LOOP] is included in the optional Reference Word Set.

/LOOP is similar to +LOOP, in that it terminates a DO loop and that it takes an incrementing value. The difference is that with /LOOP, the index and limit may range from zero to 65535, and the increment must be positive. /LOOP executes somewhat faster than +LOOP.

Number Bases

When you first load FORTH, all number conversions use base ten (decimal) for both input and output.



You can easily change the base by executing one of the following comands:

HEX	(--)	Sets the base to sixteen.
OCTAL	(--)	Sets the base to eight (available on some systems).†
DECIMAL	(--)	Returns the base to ten.

---

†For Experts

OCTAL is omitted unless the design of the particular processor compels its use.

When you change the number base, it stays changed until you change it again. So be sure to declare DECIMAL as soon as you're done with another number base.†

These commands make it easy to do number conversions in "calculator style."

For example, to convert decimal 100 into hexadecimal, enter

```
DECIMAL 100 HEX . 64 ok
```

To convert hex F into decimal (remember you are already in hex), enter

```
0F DECIMAL . 15 ok
```

Make it a habit, starting right now, to precede each hexadecimal value with a zero, as in

```
0A  0B  0F
```

This practice avoids mix-ups with such predefined words as B, D, or F in the EDITOR vocabulary.

A Handy Hint

A Definition of BINARY -- or Any-ARY

Beginners who want to see what numbers look like in binary notation may enter this definition:

```
: BINARY   2 BASE ! ;
```

The new word BINARY will operate just like OCTAL or HEX but will change the number base to two. On systems which do not have the word OCTAL, experimenters may define

```
: OCTAL   8 BASE ! ;
```

---

†For People Using Multiprogrammed Systems

When you change the number base, you change it for your terminal task only. Every terminal task uses a separate number base.

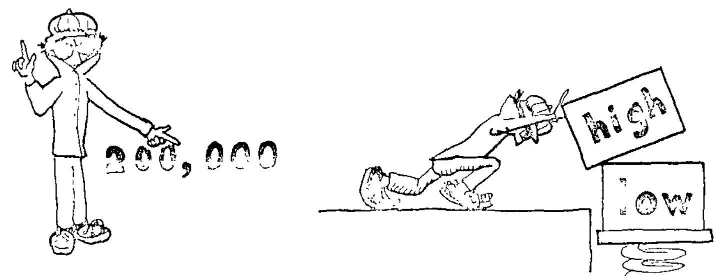
Double-length Numbers

Double-length numbers provide a range of ±2,147,483,647. Most FORTH systems support double-length numbers to some degree.†‡ Normally, the way to enter a double-length number onto the stack (whether from the keyboard or from a block) is to punctuate it with one of these five punctuation marks:

, . / - :

For example, when you type

200,000 RETURN



NUMBER recognizes the comma as a signal that this value should be converted to double-length. NUMBER then pushes the value onto the stack as two consecutive "cells" (cell is the FORTH term for sixteen bits), the high order cell on top.

---

†For polyFORTH Users:

polyFORTH includes double-length routines, but they are "electives," which means that they are written in the group of blocks which you must load each time the system is booted. This arrangement gives you the flexibility to either load these routines or delete them from your load block, according to the needs of your application.

‡FORTH-79 Standard

The Standard requires only three double-length arithmetic primitives. The optional Double Number Word Set includes many more double-length operators.

The FORTH word D. prints a double-length number without any punctuation.

D.	(d --)	Prints the signed double-length number, followed by one space.

In this book, the letter "d" stands for a double-length signed integer.

For example, having entered a double-length number, if you were now to execute D., the computer would respond:

```
D. 200000 ok
```

Notice that all of the following numbers are converted in exactly the same way:

```
12345. D. 12345 ok
123.45 D. 12345 ok
1-2345 D. 12345 ok
1/23/45 D. 12345 ok
1:23:45 D. 12345 ok
```

But this is not the same:

```
-12345
```

because this value would be converted as a negative, single-length number. (This is the only case in which a hyphen is interpreted as a minus sign and not as punctuation.)

In the next section we'll show you how to define your own equivalents to D. which will print whatever punctuation you want along with the number.

Number Formatting -- Double-length Unsigned†

$200.00 12/31/80 372-8493 6:32:59 98.6

The above numbers represent the kinds of output you can create by defining your own "number-formatting words" in FORTH. This section will show you how.

The simplest number-formatting definition we could write would be

```
: UD.  <#  #S  #>  TYPE ;
```

UD. will print an unsigned double-length number. The words <# and #> (respectively pronounced bracket-number and number-bracket) signify the beginning and the end of the number-conversion process. In this definition, the entire conversion is being performed by the single word #S (pronounced numbers). #S converts the value on the stack into ASCII characters. It will only produce as many digits as are necessary to represent the number; it will not produce leading zeroes. But it always produces at least one digit, which will be zero if the value was zero. For example:

```
12,345 UD. 12345ok
12. UD. 12ok
0 UD. 0ok
```

The word TYPE prints the characters that represent the number at your terminal. Notice that there is no space between the number and the "ok." To get a space, you would simply add the word SPACE, like this:

```
: UD.  <# #S #>  TYPE SPACE ;
```

Now let's say we have a phone number on the stack, expressed as a 32-bit unsigned integer. For example, we may have typed in

```
372-8493
```

(remember that the hyphen tells NUMBER to treat this as a double-length value). We want to define a word which will format this value back as a phone number. Let's call it .PH# (for "print the phone number") and define it thus:

---

†For Those Whose Systems Do Not Have Double-length Routines Loaded

The examples used in this and the next section won't do what you expect. The principles remain the same, however, so read these two sections carefully, then read the note on page 172.

```
: .PH#   <#  # # # # 45 HOLD  #S  #>  TYPE SPACE ;
```

Our definition of .PH# has everything that UD. has, and more. The FORTH word # (pronounced number) produces a single digit only. A number-formatting definition is reversed from the order in which the number will be printed, so the phrase

```
# # # #
```

produces the right-most four digits of the phone number.

Now it's time to insert the hyphen. Looking up the ASCII value for hyphen in the table in the beginner's section of this chapter, we find that a hyphen is represented by decimal 45. The FORTH word HOLD takes this ASCII code and inserts it into the formatted number character string.

We now have three digits left. We might use the phrase

```
# # #
```

but it's easier to simply use the word #S, which will automatically convert the rest of the number for us.

If you are more familiar with ASCII codes represented in hexadecimal form, you can use this definition instead:

```
HEX   : .PH#   <#   # # # # 2D HOLD  #S  #>  TYPE SPACE ;
DECIMAL
```

Either way, the compiled definition will be exactly the same.

Now let's format an unsigned double-length number as a date, in the following form:


```
7/15/80
```

Here is the definition:

```
: .DATE   <#  # # 47 HOLD  # # 47 HOLD  #S  #>   TYPE SPACE ;
```

Let's follow the above definition, remembering that it is written in reverse order from the output. The phrase

```
# # 47 HOLD
```

produces the right-most two digits (representing the year) and the right-most slash. The next occurrence of the same phrase produces the middle two digits (representing the day) and the left-most slash. Finally, $\boxed{\#S}$ produces the left-most two digits (representing the month).

We could have just as easily defined

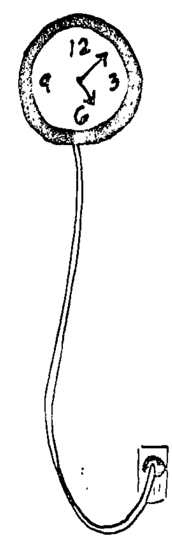
```
# # 47 HOLD
```

as its own word and used this word twice in the definition of .DATE.

Since you have control over the conversion process, you can actually convert different digits in different number bases, a feature which is useful in formatting such numbers as hours and minutes. For example, let's say that you have the time in seconds on the stack, and you want a word that will print hh:mm:ss. You might define it this way:

```
: SEXTAL   6 BASE ! ; †
: :00   #  SEXTAL #  DECIMAL  58 HOLD ;
: SEC   <#  :00 :00 #S  #>  TYPE SPACE ;
```

We will use the word :00 to format the seconds and the minutes. Both seconds and minutes are modulo-60, so the right digit can go as high as nine, but the left digit can only go up to five. Thus in the definition of :00 we convert the first digit (the one on the right) as a decimal number, then go into "sextal" (base 6) and convert the left digit. Finally, we return to decimal and insert the colon character. After :00 converts the seconds and the minutes, $\boxed{\#S}$ converts the remaining hours.



For example, if we had 4500 seconds on the stack, we would get

```
4500. SEC 1:15:00 ok
```

Table 7-2 summarizes the FORTH words that are used in number formatting. (Note the "KEY" at the bottom, which serves as a reminder of the meanings of "n," "d," etc.)

---

† For Beginners

See the Handy Hint on page 163.

TABLE 7-2 -- NUMBER FORMATTING

<# bracket-number	Begins the number conversion process. Expects an unsigned double-length number on the stack.
# number	Converts one digit and puts it into an output character string. # always produces a digit--if you're out of significant digits, you'll still get a zero for every #.
#S numbers	Converts the number until the result is zero. Always produces at least one digit (0 if the value is zero).
c HOLD	Inserts, at the current position in the character string being formatted, a character whose ASCII value is on the stack. HOLD (or a word that uses HOLD) must be used between <# and #>.
SIGN	Inserts a minus sign in the output string if the third number on the stack is negative. Usually used immediately before #> for a leading minus sign.
#> number-bracket	Completes number conversion by leaving the character count and address on the stack (these are the appropriate arguments for TYPE).

Stack effects for number formatting

phrase	stack	type of arguments
<# ... #>	(d -- adr u) or (u 0 -- adr u)	32-bit unsigned 16-bit unsigned
<# ... SIGN #>	(n $\lvert d \rvert$ -- adr u) or	32-bit signed (where n is the high-order cell of d and $\lvert d \rvert$ is the absolute value of d).
	(n $\lvert n \rvert$ 0 -- adr u)	16-bit signed (where $\lvert n \rvert$ is the absolute value).

KEY

n, n1 ...	16-bit signed numbers	adr	address
d, d1, ...	32-bit signed numbers	c	ASCII character value
u, u1, ...	16-bit unsigned numbers		

Number Formatting -- Signed and Single-length

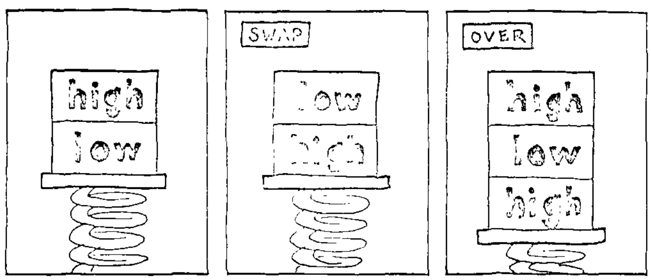
So far we have formatted only unsigned double-length numbers. The <#...#> form expects only unsigned double-length numbers, but we can use it for other types of numbers by making certain arrangements on the stack.

For instance, let's look at a simplified version of the system definition of D. (which prints a signed double-length number):

```
: D.   SWAP OVER DABS  <#  #S SIGN  #>  TYPE SPACE ;
```

The word SIGN, which must be situated within the <#...#> phrase, inserts a minus sign in the character string only if the third number on the stack is negative. So we must put a copy of the high-order cell (the one with the sign bit) at the bottom of the stack, by using the phrase

```
SWAP OVER
```



Because <# expects only unsigned double-length numbers, we must take the absolute value of our double-length signed number, with the word DABS. We now have the proper arrangement of arguments on the stack for the <#...#> phrase. The word SIGN, like HOLD, will insert the minus sign at whatever point within the character string we situate it. Since we want our minus sign to appear at the left, we include SIGN at the right of our <#...#> phrase. In some cases, such as accounting, we may want a negative number to be written

```
12345-
```

in which case we would place the word SIGN at the left side of our <#...#> phrase, like this:

```
<# SIGN #S #>
```

Let's define a word which will print a signed double-length number with a decimal point and two decimal places to the right of the decimal. Since this is the form most often used for writing dollars and cents, let's call it .$ and define it like this:

```
: .$   SWAP OVER DABS
     <#  # # 46 HOLD  #S SIGN  36 HOLD  #>  TYPE SPACE ;
```

Let's try it:

```
2000.00 .$ $2000.00 ok
```

or even

```
2,000.00 .$ $2000.00 ok
```

We recommend that you save .$, since we'll be using it in some future examples.

You can also write special formats for single-length numbers. For example, if you want to use an unsigned single-length number, simply put a zero on the stack before the word <#. This effectively changes the single-length number into a double-length number which is so small that it has nothing (zero) in the high-order cell.

To format a <u>signed</u> single-length number, again you must supply a zero as a high-order cell. But you also must leave a copy of the signed number in the third stack position for SIGN, and you must leave the absolute value of the number in the second stack position. The phrase to do all of this is

```
DUP ABS 0
```

Here are the "set-up" phrases that are needed to print various kinds of numbers:

Number to be printed	Precede `<#` by
32-bit, unsigned	(nothing needed)
31-bit, plus sign	SWAP OVER DABS (to save the sign in the third stack position for `SIGN`)
16-bit, unsigned	0 (to give a dummy high-order part)
15-bit, plus sign	DUP ABS 0 (to save the sign)

If Your System Does Not Have Double-length Routines Loaded

In this case the set-up phrases are different, as follows:

Number to be printed	Precede `<#` by
16-bit, unsigned	DUP
15-bit, plus sign	DUP ABS DUP

Even though `#` still expects two cells on the stack, in this case the significant cell must be on top (where normally the high-order cell is found). The contents of the second stack position are not used.

Double-length Operators

Here is a list of double-length math operators:† ‡

D+	(d1 d2 -- d-sum)	Adds two 32-bit numbers.	d-plus
D-	(d1 d2 -- d-diff)	Subtracts two 32-bit numbers (d1-d2).	d-minus
DNEGATE	(d -- -d)	Changes the sign of a 32-bit number.	d-negate
DABS	(d -- |d|)	Returns the absolute value of a 32-bit number.	d-absolute
DMAX	(d1 d2 -- d-max)	Returns the maximum of two 32-bit numbers.	d-max
DMIN	(d1 d2 -- d-min)	Returns the minimum of two 32-bit numbers.	d-min
D=	(d1 d2 -- f)	Returns true if d1 and d2 are equal.	d-equal
D0=	(d -- f)	Returns true if d is zero.	d-zero-equal
D<	(d1 d2 -- f)	Returns true if d1 is less than d2.	d-less-than
DU<	(ud1 ud2 -- f)	Returns true if ud1 is less than ud2. Both numbers are unsigned.	d-u-less-than
D.R	(d width --)	Prints the signed 32-bit number, right-justified within the field width.	d-dot-r

†For polyFORTH Users

The double-length routines must be loaded.

‡FORTH-79 Standard

Except for D+, D<, and DNEGATE, which are required, these words are part of the optional Double Number Word Set.

The initial "D" signifies that these operators may only be used for double-length operations, whereas the initial "2," as in [2SWAP] and [2DUP], signifies that these operators may be used either for double-length numbers or for pairs of single-length numbers.

Here's an example using [D+]:

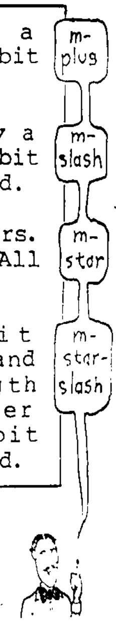
    200,000 300,000 D+ D. 500000 ok

A warning for experimenters: you can write definitions that contain double-precision operators, but you cannot include a punctuated, double-precision number inside a definition. In the next chapter we'll explain what to do instead.

Mixed-Length Operators

Here's a table of very useful FORTH words which operate on a combination of single- and double-length numbers:†

M+	(d n -- d-sum)	Adds a 32-bit number to a 16-bit number. Returns a 32-bit result.
M/	(d n -- n-quot)	Divides a 32-bit number by a 16-bit number. Returns a 16-bit result. All values are signed.
M*	(n1 n2 -- d-prod)	Multiplies two 16-bit numbers. Returns a 32-bit result. All values are signed.
M*/	(d n n -- d-result)	Multiplies a 32-bit number by a 16-bit number and divides the triple-length result by a 16-bit number (d*n/n). Returns a 32-bit result. All values are signed.



†FORTH-79 Standard

The mixed-length operators are not included in either the Required or the Double Number Word Set.

Here's an example using $\boxed{\text{M+}}$:

    200,000 7 M+ D. 200007 ok

Or, using $\boxed{\text{M*/}}$, we can redefine our earlier version of % so that it will accept a double-length argument:

    : % 100 M*/ ;

as in

    200.50 15 % D. 3007 ok

If you have loaded the definition of .$ which we gave in the last Handy Hint, you can enter

    200.50 15 % .$ $30.07 ok

We can redefine our earlier definition of R% to get a rounded double-length result, like this:

    : R% 10 M*/ 5 M+ 10 M/ ;

then

    987.65 15 R% .$ $30.08 ok

Notice that $\boxed{\text{M*/}}$ is the only ready-made FORTH word which performs multiplication on a double-length argument. To multiply 200,000 by 3, for instance, we must supply a "1" as a dummy denominator:

    200,000 3 1 M*/ D. 600000 ok

since

$$\frac{3}{1}$$

is the same as 3.

$\boxed{\text{M*/}}$ is also the only ready-made FORTH word that performs division with a double-length result. So to divide 200,000 by 4, for instance, we must supply a "1" as a dummy numerator:

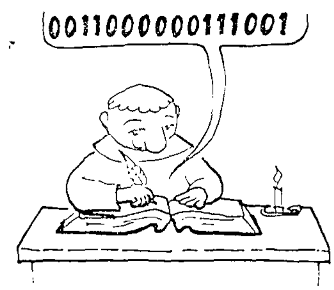
    200,000 1 4 M*/ D. 50000 ok

Numbers in Definitions

When a definition contains a number, such as

```
: SCORE-MORE   20 + ;
```

the number is compiled into the dictionary in binary form, just as it looks on the stack.



The number's binary value depends on the number base at the time you compile the definition. For example, if you were to enter

```
HEX  : SCORE-MORE  14 + ;  DECIMAL
```

the dictionary definition would contain the hex value 14, which is the same as the decimal value 20 (16 + 4). Henceforth, SCORE-MORE will always add the equivalent of decimal 20 to the value on the stack, regardless of the current number base.

If, on the other hand, you were to put the word HEX inside the definition, then you would change the number base when you execute the definition.

For example, if you were to define:

```
DECIMAL
: EXAMPLE   HEX 20 .  DECIMAL ;
```

the number would be compiled as the binary equivalent of decimal 20, since DECIMAL was current at compilation time.

At execution time, here's what happens:

```
EXAMPLE 14 ok
```

The number is output in hexadecimal.

For the record, a number that appears inside a definition is called a "literal." (Unlike the words in the rest of the definition which allude to other definitions, a number must be taken literally.)

Here is a list of the FORTH words we've covered in this chapter:

Unsigned operators

U.	(u --)	Prints the unsigned single-length number, followed by one space.
U*	(ul u2 -- ud)	Multiplies two 16-bit numbers. Returns a 32-bit result. All values are unsigned.
U/MOD	(ud ul -- u2 u3)	Divides a 32-bit by a 16-bit number. Returns a 16-bit quotient and remainder. All values are unsigned.
U<	(ul u2 -- f)	Leaves true if ul < u2, where both are treated as 16-bit unsigned integers.
DO ... /LOOP	DO: (u-limit u-index --) /LOOP: (u --)	Like DO ... +LOOP except uses an unsigned limit, index, and increment.

Number bases

HEX	(--)	Sets the base to sixteen.
OCTAL	(--)	Sets the base to eight (available on some systems).
DECIMAL	(--)	Returns the base to ten.

Number formatting operators

<#	Begins the number conversion process. Expects an <u>unsigned</u> <u>double-length</u> number on the stack.
#	Converts one digit and puts it into an output character string. $\boxed{\#}$ <u>always</u> produces a digit--if you're out of significant digits, you'll still get a zero for every $\boxed{\#}$.

#S	Converts the number until the result is zero. Always produces at least one digit (0 if the value is zero).
c HOLD	Inserts, at the current position in the character string being formatted, a character whose ASCII value is on the stack. HOLD (or a word that uses HOLD) must be used between <# and #>.
SIGN	Inserts a minus sign in the output string if the third number on the stack is negative. Usually used immediately before #> for a leading minus sign.
#>	Completes number conversion by leaving the character count and address on the stack (these are the appropriate arguments for TYPE).

Stack effects for number formatting

phrase	stack	type of arguments
<# ... #>	(d -- adr u) or (u 0 -- adr u)	32-bit unsigned 16-bit unsigned
<# ... SIGN #>	(n |d| -- adr u) or	32-bit signed (where n is the high-order cell of d and |d| is the absolute value of d).
	(n |n| 0 -- adr u)	16-bit signed (where |n| is the absolute value).

Double-length operators (Optional in FORTH-79 Standard)

D+	(d1 d2 -- d-sum)	Adds two 32-bit numbers.
D-	(d1 d2 -- d-diff)	Subtracts two 32-bit numbers (d1-d2).
DNEGATE	(d -- -d)	Changes the sign of a 32-bit number.
DABS	(d -- |d|)	Returns the absolute value of a 32-bit number.
DMAX	(d1 d2 -- d-max)	Returns the maximum of two 32-bit numbers.

DMIN	(d1 d2 -- d-min)	Returns the minimum of two 32-bit numbers.
D=	(d1 d2 -- f)	Returns true if d1 and d2 are equal.
D0=	(d -- f)	Returns true if d is zero.
D<	(d1 d2 -- f)	Returns true if d1 is less than d2.
DU<	(ud1 ud2 -- f)	Returns true if ud1 is less than ud2. Both numbers are unsigned.
DU<		Prints the signed 32-bit number, followed by one space.
D.R	(d width --)	Prints the signed 32-bit number, right-justified within the field width.

Mixed-length operators (Not required by FORTH-79 Standard)

M+	(d n -- d-sum)	Adds a 32-bit number to a 16-bit number. Returns a 32-bit result.
M/	(d n -- n-quot)	Divides a 32-bit number by a 16-bit number. Returns a 16-bit result. All values are signed.
M*	(n1 n2 -- d-prod)	Multiplies two 16-bit numbers. Returns a 32-bit result. All values are signed.
M*/	(d n n -- d-result)	Multiplies a 32-bit number by a 16-bit number and divides the triple-length result by a 16-bit number (d*n/n). Returns a 32-bit result. All values are

KEY

n, n1 ...	16-bit signed numbers	b	8-bit byte
d, d1, ...	32-bit signed numbers	f	Boolean flag
u, u1, ...	16-bit unsigned numbers	c	ASCII character value
ud, ud1, ...	32-bit unsigned numbers	adr	address

Review of Terms

Arithmetic left and right shift	the process of shifting all bits in a number, except the sign bit, to the left or right, in effect doubling or halving the number, respectively.
ASCII	a standardized system of representing input/output characters as byte values. Acronym for American Standard Code for Information Interchange. (Pronounced ask-key.)
Binary	number base 2.
Byte	the standard term for an 8-bit value.
Cell	the FORTH term for a 16-bit value.
Decimal	number base 10.
Hexadecimal	number base 16.
Literal	in general, a number or symbol which represents only itself; in FORTH, a number that appears inside a definition.
Mask	a value which can be "superimposed" over another, hiding certain bits and revealing only those bits that we are interested in.
Number formatting	the process of printing a number, usually in a special form such as 3/13/81 or $47.93.
Octal	number base 8.
Sign bit, high-order bit	the bit which, for a signed number, indicates whether it is positive or negative and, for an unsigned number, represents the bit of the highest magnitude.
Two's complement	for any number, the number of equal absolute value but opposite sign. To calculate 10 - 4, the computer first produces the two's complement of 4 (i.e., -4), then computes 10 + (-4).
Unsigned number	a number which is assumed to be positive.

Unsigned single-length number	an integer which falls within the range 0 to 65535.
Word	in FORTH, a defined dictionary entry; elsewhere, a term for a 16-bit value.

Problems -- Chapter 7

FOR BEGINNERS

1. Veronica Wainwright couldn't remember the upper limit for a signed single-length number, and she had no book to refer to, only a FORTH terminal. So she wrote a definition called N-MAX, using a BEGIN...UNTIL loop. When she executed it, she got

 32767 ok

 What was her definition?

2. Since you now know that AND and OR employ bit logic, explain why the following example *must* use [illegible] instead of +:

```
: MATCH   HUMOROUS SENSITIVE AND
    ART-LOVING MUSIC-LOVING OR  AND  SMOKING NOT  AND
      IF ." I HAVE SOMEONE YOU SHOULD MEET "  THEN ;
```

3. Write a definition that "rings" your terminal's bell three times. Make sure that there is enough of a delay between the bells so that they are distinguishable. Each time the bell rings, the word "BEEP" should appear on the terminal screen.

(Problems 4 and 5 are practice in double-length math.)

4. a. Rewrite the temperature conversion definitions which you created for the problems in Chap. 5. This time assume that the input and resulting temperatures are to be double-length signed integers which are scaled (i.e., multiplied) by ten. For example, if 10.5 degrees is entered, it is a 32-bit integer with a value of 105.

 b. Write a formatted output word named .DEG which will display a 32-bit signed integer scaled by ten as a string of digits, a decimal point, and one fractional digit.

 For example:

 12.3 .DEG RETURN 12.3 ok

Problem 4, continued

c. Solve the following conversions:

0.0° F in Centigrade
212.0° F in Centigrade
20.5° F in Centigrade
16.0° C in Fahrenheit
-40.0° C in Fahrenheit
100.0° K in Centigrade
100.0° K in Fahrenheit
233.0° K in Centigrade
233.0° K in Fahrenheit

5. a. Write a routine which evaluates the quadratic equation

$$7x^2 + 20x + 5$$

given x, and returns a double-length result.

b. How large an x will work without overflowing thirty-two bits as a signed number?

FOR EVERYONE

6. Write a word which prints the numbers 0 through 16 (decimal) in decimal, hexadecimal, and binary form in three columns. E.g.,

```
DECIMAL 0    HEX 0    BINARY     0
DECIMAL 1    HEX 1    BINARY     1
DECIMAL 2    HEX 2    BINARY    10
         ...
DECIMAL 16   HEX 10   BINARY 10000
```

7. If you enter

.. RETURN

(two periods not separated by a space) and the system responds "ok," what does this tell you?

8. Write a definition for a phone-number formatting word that will also print the area code with a slash if and only if the number includes an area code. E.g.,

```
555-1234 .PH# 555-1234 ok
213/372-8493 .PH# 213/372-8493 ok
```