DOCUMENT RESUME

ED 124 131 IR 003 519

| | |
|---|---|
| AUTHOR | Avner, Elaine |
| TITLE | PLATO User's Memo, Number Two: Basic Bit Operations. Second Edition. |
| INSTITUTION | Illinois Univ., Urbana. Computer-Based Education Lab. |
| SPONS AGENCY | Illinois Univ., Urbana.; National Science Foundation, Washington, D.C. |
| PUB DATE | Oct 75 |
| GRANT | USNSF-C-723 |
| NOTE | 31p. |
| AVAILABLE FROM | PLATO Publications, Computer-based Education Research Laboratory; 252 Engineering Research Laboratory, University of Illinois, Urbana, Illinois 61801 ($1.15, prepayment required) |
| EDRS PRICE | MF-$0.83 HC-$2.06 Plus Postage. |
| DESCRIPTORS | *Computer Assisted Instruction; *Computer Science; Computer Storage Devices; Higher Education; Manuals; *Programing |
| IDENTIFIERS | Bit Manipulation; Data Storage; PLATO IV; Programmed Logic for Automated Teaching Operations |

ABSTRACT

To help the PLATO computer-based instruction system user achieve the most efficient storage and manipulation of data, this manual begins with a review of the structure of decimal, binary, and octal number systems, and methods for converting from one system to another. The text describes the four basic operations that PLATO employs to manipulate bits of data (shifting, mask, union, and diff) and how these operations can be used to: (1) store data in an integer variable using shifting and masking; (2) pack and recover information; and (3) store data in a variable using segment. Other methods of bit manipulation are described, and an appendix provides tables of number system conversions and internal keycodes. (EMH)

ED124131

# PLATO USER'S MEMO

Number 2

Second Edition

## BASIC BIT OPERATIONS

Elaine Avner

October, 1975

CERL



Computer-based Education Research Laboratory

University of Illinois    Urbana Illinois

First Edition October 1974
Second Edition October 1975



This manuscript was prepared with partial support from the National Science Foundation (USNSF C-723) and the University of Illinois at Urbana-Champaign.

## Acknowledgment

I wish to thank Bruce Sherwood, James Ghesquiere, and William Golden for comments. Users of earlier versions of this note also made helpful suggestions. William Golden recommended distributing it as a PLATO User's Memo.

Sheila Knisley did the grueling job of typing the manuscript, and Roy Lipschutz did the drafing work.

Table of Contents

## I. Introduction

Authors often need to store large amounts of information. Use of a single variable for each piece of information would be wasteful; the information can be packed into fewer variables and retrieved when needed. Sometimes a simple yes - no or on - off or 1 - Ø is all the information desired. We shall concentrate on packing somewhat more complicated information into an integer variable (n-variable) since the bit operations described are usually valid only for integer variables.

Before we discuss bits and bit operations, let us review the three basic number systems we encounter in work with computers: decimal, binary, and octal. We shall be concerned with expressing integers in these three systems and in converting numbers from one system to another. Authors who wish to proceed directly to a discussion of "segment" may turn to page 11.

## II. Number systems

Decimal number system. The number system in everyday use is based on powers of 1Ø. The number 3842, for example, is actually 3ØØØ+8ØØ+4Ø+2, or $3\times10^3+8\times10^2+4\times10^1+2\times10^0$. The decimal (base 1Ø) system has 1Ø symbols, Ø through 9. The next integer beyond $9=9\times10^0$ (or $99=9\times10^1+9\times10^0$ or $999=9\times10^2+9\times10^1+9\times10^0$, etc.) is $1\times10^1$ (or $1\times10^2$ or $1\times10^3$, etc.). When all digits contain the highest symbol (9), integers start over again with the symbol "1" with an increase of 1 in the highest power of 1Ø. With these fundamental ideas of the maximum permitted symbol in the number system and of writing numbers in terms of powers of the base, we can use other number systems to represent any quantity. To avoid confusion, we shall indicate the base of a given number as a subscript; we write $1Ø1_{10}$ for 1Ø1 in base 1Ø or decimal; for $1Ø1_2$ for 1Ø1 in base 2 or binary; and $1Ø1_8$ for 1Ø1 in base 8 or octal.

Binary number system. The binary system (base 2) is especially convenient for computers. There are only two symbols, Ø and 1. (These may be considered off -- on switches.) Each binary digit is a bit. The number $1ØØ1111_2$ means

$$1\times2^6 + Ø\times2^5 + Ø\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^Ø .$$

In terms of the more familiar decimal system, this number is

$$1\times64 + 0\times32 + 0\times16 + 1\times8 + 1\times4 + 1\times2 + 1\times1$$

or

$$64 + 8 + 4 + 2 + 1 = 79_{10}.$$

Another way of making this last statement is $1001111_2 = 79_{10}$.

Octal number system. The octal system (base 8) uses eight symbols, 0 through 7. The octal number $4375_8$ represents

$$4\times8^3 + 3\times8^2 + 7\times8^1 + 5\times8^0,$$

or, in terms of the decimal system,

$$4\times512 + 3\times64 + 7\times8 + 5\times1,$$

or

$$2048 + 192 + 56 + 5 = 2301_{10},$$

or

$$4375_8 = 2301_{10}$$

III. Conversion from one system to another

Binary to decimal. We use the power method just described.

$$10111_2 = 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$$
$$= 1\times16 + 0\times8 + 1\times4 + 1\times2 + 1\times1$$
$$10111_2 = 23_{10}$$

Octal to decimal. Use the power method.

$$324_8 = 3\times8^2 + 2\times8^1 + 4\times8^0$$
$$= 3\times64 + 2\times8 + 4\times1$$
$$324_8 = 212_{10}$$

Decimal to binary. For small decimal integers, conversion to binary can be carried out by inspection, using the reverse of the procedure above. Any decimal integer can be broken down into the sum of powers of 2. (Appendix B. contains a table of powers of 2 up to $2^{59}$.) For example:

$$47_{10} = 32 + 8 + 4 + 2 + 1$$
$$= 2^5 + 2^3 + 2^2 + 2^1 + 2^0$$
$$= 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$$
$$47_{10} = 101111_2$$

This procedure becomes tedious for conversion of large decimal integers. Another method of conversion from decimal to binary requires only successive division of the decimal integer by 2 and recording of the remainder (which is either 0 or 1). Remainder after the first division by 2 is the lowest bit (right-most bit); remainder after division of the quotient by 2 is the next higher bit, and so on. This process is continued until the quotient is 0.

For example, convert $53_{10}$ to binary notation.

| | |
|---|---|
| $53 \div 2 = 26$, remainder 1 | 1 |
| $26 \div 2 = 13$, remainder 0 | 0 |
| $13 \div 2 = 6$, remainder 1 | 1 |
| $6 \div 2 = 3$, remainder 0 | 0 |
| $3 \div 2 = 1$, remainder 1 | 1 |
| $1 \div 2 = 0$, remainder 1 | 1 |
| | 110101 |

Thus, $53_{10} = 110101_2$

Decimal to octal. The methods given for decimal to binary conversion also apply here. Except for very small decimal integers, the powers of 8 method is cumbersome. We use the method of dividing successively by 8.

For example, convert $439_{10}$ to octal notation.

| | |
|---|---|
| $439 \div 8 = 54$, remainder 7 | 7 |
| $54 \div 8 = 6$, remainder 6 | 6 |
| $6 \div 8 = 0$, remainder 6 | 6 |
| | 667 |

Thus, $439_{10} = 667_8$.

Binary to octal. We arrange the binary number in groups of 3 bits each, starting with the right-most bit. In each group we give the bit position its binary value: $2^2$, $2^1$, or $2^0$ ( or 4, 2, or 1). If all bits in a group are set (i.e., =1), then the value of the group is 7; if none are set, the value of the group is 0. The value of the triplet is between 0 and 7, exactly the range of symbols in the octal system. When all groups have been evaluated individually and written successively, the resulting number is in octal notation. We bypass the decimal system entirely.

For example, convert $11\emptyset1\emptyset11\emptyset1\emptyset_2$ to octal notation.

1 1Ø1 Ø11 Ø1Ø

1 5 3 2

Thus, 1 1Ø1 Ø11 $\emptyset1\emptyset_2$ = $1532_8$. (Spaces may be embedded within a binary or octal number to increase readability. Such spaces are ignored during any computation.)

Octal to binary. This process is exactly the reverse of the preceding one. We take the octal number and break up each octal digit into bits. Since the greatest octal digit is 7, no more than 3 bits are needed to express one octal digit. These bits are the 4, 2, and 1 bits.

For example, convert $62451_8$ to binary notation.

6 2 4 5 1

11Ø Ø1Ø 1ØØ 1Ø1 ØØ1

Thus, $62451_8$ = 11Ø Ø1Ø 1ØØ 1Ø1 $\emptyset\emptyset1_2$.

Appendix A. contains a table of decimal, octal, and binary integers from Ø to $69_{1\emptyset}$.

IV. Bit operations on PLATO

Because of the compactness of the octal notation relative to the binary notation, the PLATO computer uses octal numbers. With practice, octal numbers can be converted to binary numbers by inspection. An octal integer is written with a small "o" preceding it to distinguish it from a decimal integer; o2765Ø3 indicates that this number is in octal notation. Each variable has 6Ø bits, so that an octal number can have up to 2Ø digits. Any preceding zeros need not be written; we can write the number above as o2765Ø3 rather than oØØØØØØØØØØØØØ2765Ø3. However, embedded and following zeros must always be included. Bits are numbered from 1 to 6Ø starting at the left end. Bit 1 is the sign bit; it has value Ø for a positive number and 1 for a negative number.

Octal representation of a negative number is the 7's complement of the representation of the positive number; that is, each octal digit is subtracted from 7. Suppose the number in decimal notation is -22. Octal representation of +22 is o26; octal representation of -22 is, therefore, o77777777777777777751. Octal representation of -Ø (negative zero) is o77777777777777777777, of -1, o77777777777777777776, and so on. Note that this procedure is equivalent to finding the 1's complement of the 6Ø-bit binary representation of the number. For example, $22_{1Ø}$ = o26 = ØØØ ØØØ...Ø1Ø $11Ø_2$. To find the 1's complement, we simply reverse the value of each bit; the result is 111 111...1Ø1 $ØØ1_2$ = o77777777777777777751 = $-22_{1Ø}$. The function "comp(x)" performs this operation. The preceding example could be written: comp(22).

<u>Shifting</u> <u>bits</u> <u>within</u> <u>a</u> <u>variable</u>. Two operations allow bits to be moved within a variable. Arithmetic right shift (ars) moves bits to the right by the specified number of bits. Bits "falling off" the right end of the octal number are lost; the sign bit (Ø or 1) is written in the vacated positions on the left end.

Example 1. o571Ø \$ars\$ 6 (Shift 6 bits to the right.)
Since the shift is 6 bits, or 2 triplet groups or 2 octal digits, we can find the result easily.
Result is o57. (The 6 bits on the right end are lost.)

Example 2. o571Ø \$ars\$ 8 (Shift 8 bits to the right.)
We first write down the number in binary notation since the shift is not by triplet groups and, therefore, not easily done by inspection. $571Ø_8$ = 1Ø1 111 ØØ1 $ØØØ_2$. We shift bits to the right by 8 positions and drop the right-most 8 bits.
Result is 1 $Ø11_2$ = $13_8$ = o13.
Hence, o571Ø \$ars\$ 8 is o13.

Example 3. o777777777777777772Ø67 \$ars\$ 6
o7777777777777777772Ø (Six bits vacated on the left are filled with the sign bit, which is 1.)

The second shifting operation is the circular left shift (cls). In contrast to the right shift, bits here are not lost. Bits falling off the left end are tacked on to the right end of the octal number, so that a bit may cycle through all positions of a variable.

Example 1. o571Ø \$cls\$ 6 (Shift 6 bits to the left.)
Since the shift is through 2 triplet groups, the result can be obtained by inspection.
Result is o571ØØØ.

Example 2. o571Ø $cls$ 8 (Shift 8 bits to the left.)

571Ø$_8$ = 1Ø1 111 ØØ1 ØØØ$_2$. Shift 8 positions to the left and fill in vacated positions on the right with bits pushed over the left end. (In this case all these bits are zeros.)

Result is 1Ø 111 1ØØ 1ØØ ØØØ ØØØ ØØØ$_2$ = 2744ØØØ$_8$ = o2744ØØØ.

Hence, o571Ø $cls$ 8 is o2744ØØØ.

Example 3. o77777777777777772Ø67 $cls$ 6.

By inspection, result is o77777777777777772Ø6777.

The 6 bits pushed off the left side are tacked on to the right side.

Example 4. o77777777777777772Ø67 $cls$ 8

Result is o777777777777775Ø33777.

(Write down the bits, shift, and regroup.)

Example 5. o12347654ØØØØØØØØØØØØØ $cls$ 12

The shift is to the left through 4 triplets.

Result is o7654ØØØØØØØØØØØØØ1234.

<u>Mask</u>. The mask operations allows only certain bits to be "seen." The mask is analogous to the intersection of two sets. In the form n3⇐ n1 $mask$ n2, bits must be set (=1) in <u>both</u> n1 and n2 for the corresponding bits to be set in n3. There are four possible combinations for the bits, as shown in the following table.

| n1 | n2 | n3⇐ n1 $mask$ n2 |
|---|---|---|
| Ø | Ø | Ø |
| 1 | Ø | Ø |
| Ø | 1 | Ø |
| 1 | 1 | 1 |

Example 1. The result of masking the binary number 1ØØ 111$_2$ with 11Ø ØØ1$_2$ is 1ØØ ØØ1$_2$. In octal notation we have:

o47 $mask$ o61, which results in o41.

Example 2. Mask the binary number 1ØØ 111$_2$ with 11$_2$; result is 11$_2$.

In octal notation this problem reads:

o47 $mask$ o3, which results in o3.

The mask can act as a "window" on a variable. When bits are set in n2, the bits of n1 are visible; when bits are not set in n2, the window is closed and bits of n1 are not visible. The masking operation does not change the relative positions of bits.

Example 3. Suppose you wish to see the right-most 8 bits (bits 53 through 6Ø) of variable n1. Set up a mask in which only these bits are set, i.e., $11\ 111\ 111_2$ = o377
Required expression is n1 $mask$ o377.

Example 4. Suppose in addition to bits 53 through 6Ø you also wish to see bits 39, 4Ø, 41, 46, 47, and 48. Set up the mask
$1\ 11Ø\ ØØØ\ 111\ ØØØ\ Ø11\ 111\ 111_2$ = o16Ø7Ø377.
Required expression is n1 $mask$ o16Ø7Ø377.

Example 5. Suppose you wish to see only bits 39, 4Ø, 41, 46, 47, and 48. Mask is then $1\ 11Ø\ ØØØ\ 111\ ØØØ\ ØØØ\ ØØØ\ ØØØ_2$ = o16Ø7ØØØØ.
Required expression is n1 $mask$ o16Ø7ØØØØ.

The TUTOR functions "lmask(x)" and "rmask(x)" set up special numbers which may be used in bit operations. The function "lmask(48)" gives a left-justified octal number of 48 bits; that is, the left-most 48 bits of the number are set. The function "rmask(8)" gives a right-justified number of [illegible] bits, where the right-most 8 bits are set. Thus, example 3 above could have been written: n1 $mask$ rmask(8).

<u>Union</u>. The union operation is analogous to the union of two sets. In the form n3⇐ n1 $union$ n2, if a bit is set in <u>either</u> n1 or n2 or in both, then the corresponding bit is set in n3. The following table gives the possible combinations.

| n1 | n2 | n3⇐ n1 $union$ n2 |
|---|---|---|
| Ø | Ø | Ø |
| 1 | Ø | 1 |
| Ø | 1 | 1 |
| 1 | 1 | 1 |

Example 1. Find the union of the binary numbers $1ØØ\ 111_2$ and $11Ø\ ØØ1_2$. Result is $11Ø\ 111_2$. In octal notation the problem reads: o47 $union$ o61, which results in o67.

Example 2. Find the union of $1ØØ111_2$ with $11_2$. Result is $1ØØ\ 111_2$. (Remember the preceding zeros in the second binary number.) In octal notation: o47 $union$ o3 results in o47.

The union operation is useful in setting specific bits in a variable.

Example 3. Suppose you want to be sure that bits 53 through 6Ø of a variable, n1, are set, independent of the other bits. Set the new value of the variable equal to the union of the old value and 11 111 111$_2$ or o377 (or rmask(8)).

n1 ⇐ n1 $union$ o377 or n1 ⇐ n1 $union$ rmask(8)

Example 4. Variables may also be used in all bit operations. For example, n1 $union$ n4 is a legitimate expression.

__Diff__. In the form n3⇐ n1 $diff$ n2, a bit is set in n3 only if corresponding bits are set in __either__ n1 or n2 but __not in both__. If corresponding bits in n1 and n2 are different, then the corresponding bit is set in n3. If corresponding bits in n1 and n2 are the same, the corresponding bit is not set in n2. The possible combinations are shown below.

| n1 | n2 | n3⇐ n1 $diff$ n2 |
|---|---|---|
| Ø | Ø | Ø |
| 1 | Ø | 1 |
| Ø | 1 | 1 |
| 1 | 1 | Ø |

As the table shows, if the bit in n2 is set, then the corresponding bit in n3 is the complement of the corresponding bit in n1 (set bits are unset and vice versa). If the bit in n2 is not set, the corresponding bit in n3 is the same as the corresponding bit in n1.

Example 1. Use the diff operation on the binary numbers 1ØØ 111$_2$ and 11Ø ØØ1$_2$. Result is Ø1Ø 11Ø$_2$. In octal notation the problem reads:

o47 $diff$ o61, which results in o26.

Example 2. Use the diff operation on the binary numbers 1ØØ 111$_2$ and 11$_2$. Result is 1ØØ 1ØØ$_2$. (Again, remember the preceding zeros in the second binary number.) In octal notation, we have o47 $diff$ o3, which results in o44.

The diff operation is useful in reversing specific bits in a variable.

Example 3. Suppose you wish to reverse bits 53 through 60 of variable n1 and set the result to n3. Take the diff with $11\ 111\ 111_2$ = o377. (The preceding 17 octal zeros will not cause any changes in the bits of n1.)

n3 ⇐ n1 \$diff\$ o377 or n3 ⇐ n1 \$diff\$ rmask(8)

Example 4. Reverse bits 53 through 60 and also bits 45, 46, 48, 49, and 50 of variable n1 and set the result to n4. You need $1\ 101\ 110\ 011\ 111\ 111\ 111_2$ = o156377.

n4 ⇐ n1 \$diff\$ o156377

Combination of operations. Suppose the value of bits 31, 32, 33 and 34 of n1 are needed. We must adjust the mask so only these bits come through. Mask is 111 100 000 000 000 000 000 000 000 000. Thus n3 ⇐ n1 \$mask\$ o7400000000. We may also shift the bits to the right end before operating with the mask.

n4 ⇐ (n1 \$ars\$ 26) \$mask\$ o17

or

n5 ⇐ (n1 \$ars\$ 24) \$mask\$ o74

or

n5 ⇐ (n1 \$cls\$ 36) \$mask\$ o74

(These shifts preserve the original triplets. Result for n5 is the same in the two examples.)

If the only information needed is the values of the bits, then these operations are adequate. The results for n3, n4, and n5 are not equal, however, since the shifts are of different amounts, and the mask does not change the positions of the bits. Then

n4 , n3 \$ars\$ 26 , n5 \$ars\$ 2 are equivalent;
n3 , n4 \$cls\$ 26 , n5 \$cls\$ 24 are equivalent;
n5 , n3 \$ars\$ 24 , n4 \$cls\$ 2 are equivalent.

## V. Applications

Storing data in an integer variable using shifting and masking. Suppose information to be stored in an integer variable consists of (in base 10): (1) an integer ranging from 0 to 50; (2) a fractional number ranging from

-3.∅ to +3.∅, given to ∅.1; (3) an integer ranging from -1∅ to +5. These may be packed into the variable in any order. Let us consider them in sequence.

(1) ∅ to 5∅ This number is a positive integer and requires no scaling. Consider the maximum number of bits required to pack it into a variable. $5\emptyset_{1\emptyset} = 62_8 = 110\ \emptyset 1\emptyset_2$. This number will require no more than 6 bits.

(2) -3.∅ to +3.∅ This number requires scaling, both for sign and fractional part, since this method does not provide for storing signed numbers or fractions. After addition of 3.∅ to eliminate negative values and multiplication by 1∅ to eliminate the fractional part, the maximum value is 6∅. $6\emptyset_{1\emptyset} = 74_8 = 111\ 1\emptyset\emptyset_2$. The scaled value will require no more than 6 bits. The original value may be recovered in the unpacking.

(3) -1∅ to +5 This number requires scaling to eliminate negative values. After addition of 1∅, the maximum value is 15. $15_{1\emptyset} = 17_8 = 1\ 111_2$. This scaled number will require no more than 4 bits. Again, the original value may be recovered in unpacking.

<u>Packing and recovering the information</u>. Suppose the information is stored in variables n1, v2, and n3, and that it is to be packed into variable n1∅∅, for example. The recovered information will be in variables defined earlier as a, b, and c, respectively. The code might be written in the following way:

```
*        ∅ ≤ n1 ≤ 5∅,  -3.∅ ≤ v2 ≤ +3.∅,  -1∅ ≤ n3 ≤ +5
*        to pack:
define   a=v91,b=v92,c=v93
*
calc     n2⇐ 1∅(v2+3)          $$ scale for sign and decimal
         n3⇐ n3 + 1∅           $$ scale for sign
         n1∅∅⇐ n1 + (n2 $cls$ 6) + (n3 $cls$ 12)
*        n1 in right-most 6 bits; cls n2 over 6 bits and store in next
*        6 bits; cls n3 over 12 bits and store in next 4 bits
*        to recover:
calc     a⇐ n1∅∅ $mask$ o77       $$ look at right-most 6 bits
         b⇐ ((n1∅∅ $ars$ 6)$mask$ o77)/1∅ - 3     $$ shift, mask, scale
         c⇐ ((n1∅∅ $ars$ 12)$mask$ o17) - 1∅      $$ shift, mask, scale
```

The last two lines may also be written

```
        calc    b⇐ ((n1ØØ $mask$ o77ØØ)$ars$ 6)/1Ø - 3    $$ mask first
                c⇐ ((n1ØØ $mask$ o17ØØØØ)$ars$ 12) - 1Ø
```

To check that all is working well, you might add the following code:

```
        show    a
        showt   b,5,1   } or    write  ⊲s,a⊳  ⊲s,b⊳  ⊲s,c⊳
        showt   c,5
```

This procedure becomes economical when large amounts of data of similar formats are to be packed into many variables.

We shall now discuss the problem of packing using the segment feature, which performs many of the operations automatically.

<u>Storing data in a variable using segment</u>. The segment feature allows successive variables to be broken up into segments or bytes for purposes of storing data. The author selects a byte size which will accommodate the numbers. To use the segment feature, a statement must be added to the set of definitions with the following form:

```
        define  defns
                segment,name=starting variable,number of bits per byte,s
        *       put no other definitions on the same line with segment
        *       definition
        *       last argument, s, is optional, and is needed for signed
        *       numbers.
```

The byte size may be from 1 to 59 and cannot be a variable. The address of the starting variable also may not be a variable. (The address of the variable is the number attached to it; e.g., the address of v59 is 59; address of n13 is 13. Hence, one may not use v(v48) in the segment definition.) Unlike the example discussed in the preceding section, v-variables or n-variables may be used with the segment feature. However, only integer data may be stored. When given, the last argument in the segment definition indicates that storage of negative as well as positive integers is permitted. Without this argument only positive integers may be packed correctly. If the last argument is included, byte size must be increased by 1 to allow one bit for the sign. The convention for octal representation of negative integers is then similar to that discussed at the top of page 5.

Segmenting starts at the left end of the variable. If the byte size does not subdivide the 60 bits evenly, the extra bits at the right end of each segmented variable are unused. For example, a variable can contain seven 8-bit segments with 4 bits left over.

To fit into an <u>unsigned</u> segment of size n bits, integers may range from 0 to $2^n-1$. For a <u>signed</u> segment of size n bits, integers may range from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$. Attempts to fit integers outside these ranges will give erroneous results. Suppose we have a byte size of 7. In the table of powers of 2 in Appendix B., we find that such a segment can store positive integers from 0 to 127 or signed integers from -63 to +63.

More often you have a batch of data with a known maximum absolute value and wish to select the byte size. If this value is such that $2^{n-1} \le |\text{maximum}| < 2^n$, then the byte size for unsigned integers is n and for signed integers, n+1. (|maximum| indicates maximum absolute value; i.e., consider only the value and disregard the sign.) Suppose we want to fit the integer 17689 into a segment. We find from Appendix B. that this integer lies between $2^{14}$ and $2^{15}$. For an unsigned segment the byte size must be at least 15. For a signed segment we must allow a byte size of at least 16. If the integer is -17689, the procedure is the same, except that we are restricted to signed segments; the byte size must be at least 16.

Suppose we want to break up variables into 6-bit bytes and need 100 such bytes. Then 10 variables will be required (ten 6-bit bytes per variable). Suppose also that some of the data to be packed consist of negative numbers. We might store some of the information as follows:

```
define  seta
        segment,class=v10,6,s
*       integers from -31₁₀ to +31₁₀ may be packed without loss of
*       information
zero    v10,10              $$ initialize v10 through v19
calc    v1 ⇐ 1.5
        n2 ⇐ 4
        class(1) ⇐ 2        $$ set 1st byte to 2
        class(2) ⇐ 3        $$ set 2nd byte to 3
        class(4) ⇐ 7        $$ set 4th byte to 7       etc.
        class(20) ⇐ class(1) + class(4)
        class(40) ⇐ class(20) + 10
        class(45) ⇐ -8
        class(12) ⇐ n2
        class(32) ⇐ 10v1
```

As a check the packed information may be displayed:

```
show    class(4)
showt   class(32)/1Ø,4,1
```

Values may also be displayed by

```
write   ⊲s,class(4)⊳  ⊲s,class(32)/1Ø⊳
```

The calculations above result in the following non-zero variables:

$$v1\emptyset \text{ is o } \underbrace{\emptyset 2}_{class(1)}\ \overbrace{\emptyset 3}^{class(2)}\ \emptyset\emptyset\ \underbrace{\emptyset 7}_{class(4)}\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset$$

$$v11 \text{ is o } \emptyset\emptyset\ \underbrace{\emptyset 4}_{class(12)}\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \underbrace{11}_{class(2\emptyset)} \qquad (11_8 = 9_{1\emptyset})$$

$$v13 \text{ is o } \emptyset\emptyset\ \underbrace{17}_{class(32)}\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \underbrace{23}_{class(4\emptyset)} \qquad (17_8 = 15_{1\emptyset},\ 23_8 = 19_{1\emptyset})$$

$$v14 \text{ is o } \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \underbrace{67}_{class(45)}\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset\ \emptyset\emptyset \qquad (\text{here; } 67_8 = -8_{1\emptyset})$$

Preceding zeros are included only for ease in counting bytes and need not usually be included in writing the octal numbers.

Packing of the negative number may be analyzed further. Consider only the relevant byte of v14.

$$o67 = 67_8 = 11\emptyset\ 111_2$$

Since, according to our segment definition, we are dealing with a signed number, the left-most bit is interpreted as the sign bit, and since it is 1, the number is negative. To obtain this number we take the 1's complement of the binary number; we flip the bits of $11\emptyset\ 111_2$ and obtain $\emptyset\emptyset 1\ \emptyset\emptyset\emptyset_2 = 1\emptyset_8 = 8_{1\emptyset}$. The number stored in this byte is the negative of this integer, or $-8_{1\emptyset}$

Consider the example given previously. Information to be stored consists of (1) an integer ranging from Ø to 5Ø, represented by n1; (2) a number ranging from +3.Ø to +3.Ø represented by v2; (3) an integer ranging from -1Ø to +5, represented by n3.

The integer stored in n1 requires a maximum of 6 bits. Since the sign argument in the segment must be included to allow for negative numbers in the other data, an additional bit must be used, so the byte size is 7.

The number stored in v2 requires scaling for the fractional part but not for the sign as previously. Scaled values range from -30 to +30. Since $30_{10} = 36_8 = 11\ 110_2$ and one additional bit must be allowed for the negative values, this byte requires 6 bits.

The integer stored in n3 does not require scaling for sign as before. The largest absolute value that must be considered is $10_{10}$. Since $10_{10} = 12_8 = 01\ 010_2$, and $-10_{10} = 10\ 101_2$ here, this byte requires 5 bits.

For simplicity let all three bytes be size 7. Then there are 8 segments to a variable, with 4 bits left over at the right end. Suppose the data are stored starting in v100. The code might be written:

```
*        0 ≤ n1 ≤ 50;  -3.0 ≤ v2 ≤ +3.0;  -10 ≤ n3 ≤ +5
*
define   segs
         segment,info=v100,7,s
*
calc     info(1) ⇐ n1         $$ n1, v2, n3 have been previously set
         info(2) ⇐ 10v2       $$ scale v2 for fractional part
         info(3) ⇐ n3
*
*        check values by displaying:
*
show     info(1)
showt    info(2)/10,3,1       $$ scale back
showt    info(3),4
```

Values may also be displayed by

```
write   ◁s,info(1)▷  ◁s,info(2)/10▷  ◁s,info(3)▷
```

Note the difference in location in the variable from the earlier example; there the information was packed starting at the right end of the variable, while with segment the packing starts at the left end.

The same variable(s) can be segmented into bytes of different sizes. For example, suppose three 9-bit bytes and four 1-bit bytes are needed. The -define- could be written as follows:

```
define  packup
        segment,numb=v100,9,s
        segment,onoff=v100,1
        mean=numb(1),sd=numb(2),coef=numb(3)
        proba=onoff(28),probb=onoff(29),probc=onoff(30),probd=onoff(31)
```

Note that segmented values may be referenced in subsequent defined variables. Any permutation of the segmented data could have been used. In using this multiple segment care must be taken not to overwrite one segment with another. The three segments above, "mean," "sd," and "coef," each require nine bits. Since they are defined sequentially, they occupy bits 1 through 27. Hence in defining or using the next group of segments, "proba," "probb," "probc," and "probd," we must not use any of these first 27 bits; the first bit available is number 28.

The segment feature divides successive variables into segments starting at the left end of the first variable and proceeding "horizontally" across variables. The first and second segments, for example, are adjacent segments in the first variable. Another form of segment is the vertical segment, which behaves exactly as its name implies. It breaks up successive variables into segments and proceeds "vertically" across variables. With vertical segmenting successive segments are in successive variables; the first and second segments are in the first and second variables. The format for the definition of the vertical segment is:

```
define  defns
        segment,vertical,name=starting variable,starting bit
           position,number of bits per byte,s
*       last argument is needed only if signed numbers are used
```

Suppose the following information is to be stored in vertical segments: (1) integers which range from Ø to 5×1Ø$^{11}$; (2) integers which range from -1248 to +892; (3) integers which range from Ø to 135. The byte sizes for these values are, respectively, 39 bits, 12 bits, and 8 bits. (Refer to the table in Appendix B.)

The definitions for the vertical segments in this example could be written:

```
define  all
        segment,vertical,big=v22,1,39
        segment,vertical,neg=v22,4Ø,12,s
        segment,vertical,small=v22,52,8
```

Suppose the following calculations are performed:

```
calc    big(1) ⇐ 1000
        big(2) ⇐ 10^9
        neg(1) ⇐ 890
        neg(2) ⇐ -1200
        small(1) ⇐ 132
        small(2) ⇐ 15
```

These segments are all contained in variables v22 and v23. The calculations result in the following values for these variables.

v22 is o0000000001750 1572 410.

v23 is o00007346545000 5517 036.

Consider v22:

first 39 bits: o1750 = $1000_{10}$ (which is big(1))

next 12 bits: o1572 = $890_{10}$ (which is neg(1); sign bit is 0)

next 8 bits: o410$ars$1 = o204 = $132_{10}$ (which is small(1); the shift of 1 bit is helpful in evaluating the first 8 of the remaining 9 bits)

Consider v23:

first 39 bits: o7346545000 = $10^9_{10}$ (which is big(2))

next 12 bits: o5517 = -(o2260) = $-1200_{10}$ (which is neg(2); sign bit is 1)

next 8 bits: o036$ars$1 = o17 = $15_{10}$ (which is small(2))

## VI. Other methods of bit and character manipulation

Some TUTOR commands and functions allow bit and character manipulation. These features are summarized below. The TUTOR Language by Bruce Sherwood and lesson "aids" provide details and exceptions.

A character requires 6 bits, so that a variable can contain up to 10 characters. Appendix C. contains a table of internal keycodes, numerical representations, of more commonly used characters. In general, use of the keycodes in this table should be restricted to inspection of the contents of a variable. Suppose the expression 5+3=2×4 is stored in v1 with a -storea- command. The octal representation of v1 is o40453654356437000000. To specify characters in command tags, write the characters in single quotes (') if characters are left-justified (stored in the left end of the variable) or in double quotes (") if characters are right-justified (stored in the right end of the variable). For example, 'qed' occupies the first three characters (or the first six octal digits or the first 18 bits) of a variable; "qed" occupies the last three characters (or the last six octal digits or the last 18 bits) of a variable. In terms of the numerical codes from Appendix C., 'qed' is o21Ø5Ø4ØØØØØØØØØØØØØØ and "qed" is oØØØØØØØØØØØØØØ21Ø5Ø4 or simply o21Ø5Ø4. In the -search- command (discussed below), it is far easier to interpret 'qed' than o21Ø5Ø4ØØØØØØØØØØØØØØ, although both expressions have the same meaning. Use of quotes with the characters also avoids the problem of rewriting if the numerical codes should be changed in the future.

In all commands below involving character manipulation, character positions are numbered from the left, i.e., the left-most position is 1; the right-most, 1Ø. Except for -find- and -findall-, these commands may use either v- or n-variables. The -find- and -findall- commands must use n-variables.

-pack-.

This command packs a character string into variable(s) starting at the left end (character position 1) of the indicated starting variable. Any unused characters to the right are filled with octal zeros.

```
pack     v1,string          $$ puts character string starting in v1
showa    v1                 $$ displays string (up to 1Ø characters)
*
pack     v5,v6,string       $$ puts character string starting in v6
                               and character count in v5
showa    v6,v5              $$ displays string
```

(When packing up to 1Ø characters with the -calc- command, calc n1⇐ "string" packs the string into the variable at the right end; calc n1⇐ 'string' packs the string at the left end. Unused characters are filled with octal zeros. The -storea- command packs a response into the left end of the variable with octal zeros filling out unused character positions.)

-move-.

This command moves characters from one character string to another.

```
move     from starting variable,from starting position,to starting
         variable,to starting position,no. of characters (optional)

e.g.,

move     v1,7,v8,11,2
```

This statement causes 2 characters starting at character position 7 of the string starting in v1 to be moved to position 11 of the string starting in v8, overwriting characters already at that position. If the number of characters is omitted, it is assumed to be 1.

Another version of the -move- command uses a character string as an argument. For example,

```
move    'plato',4,v8,10
```

moves the character in position 4 of the string 'plato' (t, code=o24) to position 10 of the string starting in v8. If v8 was previously zeroed, it now has the value o24. More than one character may be moved. In

```
move    'plato',4,v8,8,2
```

the two characters starting in position 4 (to, code=o2417) are moved to position 8 of v8. If v8 was previously zeroed, it is now o241700.

-itoa-.

This command converts an integer to an alphameric string.

```
itoa    variable where integer is stored,variable where alphameric
        string is stored (left-justified),variable where number of
        characters is stored (optional)

e.g.,

calc    n1 ⇐ 24680
itoa    n1,n6,n5
```

This statement causes the integer stored in n1, 24680, to be converted to an alphameric string which is stored in n6; n6 in octal format is now o35374143330000000000. (See Appendix C., internal keycodes.) Since the third argument is included in this example, n5 contains the number of characters, 5. If the integer stored in n1 is -24680, then n6 contains o4635374143330000000 and n5 contains 6.

-search-.

This command scans a character string for an object character string. The scanned string is not broken up into words, or variables. The scan can be considered as "horizontal" across a character string, which may extend over several variables:

```
        scan ——→
        v1,v2,v3,v4,...
```

-search-. (6 arguments)

This version of -search- looks for the first occurrence of a character string.

```
search  object of search (left-justified),length of object-string,
        (≤1Ø characters),starting variable of string to search,
        number of characters in string to search,relative character
        position at which to start search,variable where relative
        character position of object string is stored

e.g.,
storea  v3Ø,jcount               $$ student types:  cats + dogs = animals.
ok
search  '=',1,v3Ø,jcount,1,v1
```

In this example the student's response is of length jcount (21 in this case, including embedded spaces) and is stored starting at variable v3Ø, with 1Ø characters per variable. This string is searched for the object, =, which is of length 1. The search starts at variable v3Ø at the first character, relative character position 1. After the search is completed, the variable v1 contains 13 since the object string starts at the 13th character position.

Another example using the same string but with different search parameters is

```
search  'cat',3,v3Ø,jcount,2,v1
```

The search now starts at character position 2. After the search is completed, v1 contains -1 since the object string, cat, starts at character position 1 and was not found.

-search-. (7 arguments)

This version of -search- looks for the specified number of occurrences of a character string.

```
search  object of search (left-justified),length of object string
        (≤1Ø characters),starting variable of string to search,number
        of characters in string to search,relative character position
        at which to start search,variable for storing number of times
        object string is found,number of following variables for
        storing relative positions of object string.
```

The string in the previous examples, i.e., cats + dogs = animals, is searched for the character s.

```
search  's',1,v3Ø,jcount,1,v1,jcount   $$ will store all possible
                                       $$ occurrences of object string
```

After the search the following values are contained in variables v1 through v(jcount):

| | | | |
|---|---|---|---|
| v1 | equals | 3 | (number of occurrences of object string) |
| v2 | " | 4 | (relative character position of 1st occurrence) |
| v3 | " | 11 | (relative character position of 2nd occurrence) |
| v4 | " | 21 | (relative character position of 3rd occurrence) |
| v5 | " | -1 | (no further occurrences of object string) |

v6 through v(jcount) are unchanged.

-find-.

This command scans a set of variables for the first occurrence of an object bit pattern. Each variable (with or without a mask) is compared with an object bit pattern (with or without the same mask). The scan can be considered "vertical," with each variable scanned independently:

```
scan  n1
scan  n2
scan  n3
 .    .
 .    .
 .    .
```

find    object bit pattern,integer variable at which to start search, number of variables,integer variable for relative found location,increment between variables (optional),mask (optional)

e.g.,

find    'we',n5,26,n1ØØ

This statement causes a scan of each of 26 variables from n5 through n3Ø for the first occurrence of the bit pattern, 'we', which is o27Ø5ØØØØØØØØØØØØØØØ. Since the increment between variables is omitted, it is set to 1, and since the mask is omitted the entire variable is compared with the entire bit pattern. If the pattern is matched, n1ØØ is set to the relative position of the variable (e.g., to Ø if match is in n5, to 2 if match is in n7, to 25 if match is in n3Ø). If no match is found, n1ØØ is set to -1.

The bit pattern of interest is not restricted to being left-justified or right-justified. Suppose the relevant information is stored in segmented variables in the fourth 12-bit segment. The variables would then look like: XXXXXXXXXXXXQQQQXXXX, where the Q's represent the segment of interest and the

X's can be any value. Suppose, also, that we are no longer comparing the entire variable with the entire object bit pattern so we need the mask. We could have a statement like

```
find    "we"$cls$12,n5,26,n1ØØ,1,o7777ØØØØ
```

or

```
find    "wexx",n5,26,n1ØØ,1,o7777ØØØØ
```

(Here, x can be any 6-bit character.) Since a mask is specified, the increment must be given to avoid ambiguity.

Of course, a numerical bit pattern is also acceptable as an argument. (However, see page 17 and Appendix C. for a precaution against use of octal representations of keycodes.) For example,

```
find    Ø,n5,26,n1ØØ
```

returns the relative position of the first variable which matches the object bit pattern and which, therefore, has all bits set to Ø since no mask is used. (Note: The object bit pattern should not be written with quotes, since "Ø" means the internal code corresponding to the character Ø, which is o33.)

-findall-.

The -findall- command is similar to the -find- command, but it picks up as many occurrences of the object bit pattern as desired.

```
findall object bit pattern,integer variable at which to start search,
        number of variables to search,integer variable where found
        count is stored,number of following variables where relative
        found locations of pattern are stored,increment between
        variables (optional),mask (optional)
```

e.g.,

```
findall 'we',n5,26,n1ØØ,6,5
```

In this example, the scan for the object bit pattern is specified only for every fifth variable: n5, n1Ø, n15, n2Ø, n25, and n3Ø. The number of found locations desired is six, so that all possible locations will be stored. Since the increment is not equal to 1, it must be specified. Since the mask is omitted, the entire variable is compared with the object bit pattern.

Suppose variables n5, n2ø and n3ø contain the object bit pattern. After -findall- is executed, the following values are contained in variables n1øø through n1ø6:

```
     n1øø equals 3     (number of occurrences of object bit pattern)
     n1ø1   "    ø     (relative position of 1st occurrence: n5)
     n1ø2   "    15    (relative position of 2nd occurrence:  n2ø)
     n1ø3   "    25    (relative position of 3rd occurrence:  n3ø)
     n1ø4   "    -1    (no further occurrences of object bit pattern)
     n1ø5 and n1ø6 are unchanged
```

With both -find- and -findall- a negative increment causes a backwards pass through the list. However, relative position is still counted from the first variable in the list. For example, suppose in the list below the number 483 is in variables n2, n4, and n5. The forward and backward scans with -find- produce the found locations indicated.

```
     find    483,n1,6,n1øø              find    483,n1,6,n1øø,-1

               scan  n1                           scan  n1
             | scan  n2                         ^ scan  n2
     forward | scan  n3                backward | scan  n3
             | scan  n4                         | scan  n4
             v scan  n5                         | scan  n5
               scan  n6                           scan  n6

     n1øø is 1 (first occurrence        n1øø is 4  (first occurrence
                in n2)                              in n5)
```

bitcnt(x)

The TUTOR function "bitcnt(x)" counts the number of bits set to 1 in its argument. For example:

```
     bitcnt(58) = bitcnt(o72) = 4
     bitcnt(87) = bitcnt(o127) = 5
     bitcnt(132) = bitcnt(o2ø4) = 2
     bitcnt(o655$mask$o263) = bitcnt(o241) = 3
     bitcnt(-5) = bitcnt(o77777777777777777772) = 58
```

The argument may be a variable. In most cases an integer variable would be used for the argument. For example:

```
     *
     calc    n1⇐ 12           $$ in octal notation n1 is o14
             v1ø⇐ bitcnt(n1)  $$ value of v1ø is 2
     *
```

However,

```
*
  calc    v2⇐ 12           $$ in octal notation v2 is o17236ØØØØØØØØØØØØØØ
          v11⇐ bitcnt(v2) $$ value of v11 is 9 .
```

(See The TUTOR Language, by Bruce Sherwood, chapter 9 for a discussion of octal representation of v-variables.)

comp(x).

The function "comp(x)" was mentioned on page 5. It reverses the value of the bits in its argument so that set bits are unset and vice versa. For example:

comp(58) = comp(o72) = o777777777777777777Ø5

comp(-82) = comp(o77777777777777777655) = o122

lmask(x), rmask(x).

The functions "lmask(x)" and "rmask(x)" were discussed on page 7. These functions establish octal numbers with either left-most or right-most bits set. The argument determines how many bits are set. For example:

lmask(48) = o7777777777777777ØØØØ

lmask(14) = o77776ØØØØØØØØØØØØØØØ

rmask(14) = oØØØØØØØØØØØØØØØ37777

zbpw, zbpc, zcpw.

The system variable "zbpw" is the number of bits per TUTOR word (or variable). It is equal to 6Ø. The system variable "zbpc" is the number of bits per character and equals 6. The system variable "zcpw" is the number of characters per TUTOR word (or variable) and equals 1Ø.

These system variables and functions may be combined. For example, rmask(zbpc), comp(lmask(zbpc)), and bitcnt(comp(n1)) are all legal.

Appendix A.

Decimal, Octal, and Binary Numbers from Ø to $69_{10}$

| Decimal | Octal | Binary | Decimal | Octal | Binary |
|---|---|---|---|---|---|
| Ø | Ø | Ø ØØØ ØØØ | 35 | 43 | Ø 1ØØ Ø11 |
| 1 | 1 | Ø ØØØ ØØ1 | 36 | 44 | Ø 1ØØ 1ØØ |
| 2 | 2 | Ø ØØØ Ø1Ø | 37 | 45 | Ø 1ØØ 1Ø1 |
| 3 | 3 | Ø ØØØ Ø11 | 38 | 46 | Ø 1ØØ 11Ø |
| 4 | 4 | Ø ØØØ 1ØØ | 39 | 47 | Ø 1ØØ 111 |
| 5 | 5 | Ø ØØØ 1Ø1 | 4Ø | 5Ø | Ø 1Ø1 ØØØ |
| 6 | 6 | Ø ØØØ 11Ø | 41 | 51 | Ø 1Ø1 ØØ1 |
| 7 | 7 | Ø ØØØ 111 | 42 | 52 | Ø 1Ø1 Ø1Ø |
| 8 | 1Ø | Ø ØØ1 ØØØ | 43 | 53 | Ø 1Ø1 Ø11 |
| 9 | 11 | Ø ØØ1 ØØ1 | 44 | 54 | Ø 1Ø1 1ØØ |
| 1Ø | 12 | Ø ØØ1 Ø1Ø | 45 | 55 | Ø 1Ø1 1Ø1 |
| 11 | 13 | Ø ØØ1 Ø11 | 46 | 56 | Ø 1Ø1 11Ø |
| 12 | 14 | Ø ØØ1 1ØØ | 47 | 57 | Ø 1Ø1 111 |
| 13 | 15 | Ø ØØ1 1Ø1 | 48 | 6Ø | Ø 11Ø ØØØ |
| 14 | 16 | Ø ØØ1 11Ø | 49 | 61 | Ø 11Ø ØØ1 |
| 15 | 17 | Ø ØØ1 111 | 5Ø | 62 | Ø 11Ø Ø1Ø |
| 16 | 2Ø | Ø Ø1Ø ØØØ | 51 | 63 | Ø 11Ø Ø11 |
| 17 | 21 | Ø Ø1Ø ØØ1 | 52 | 64 | Ø 11Ø 1ØØ |
| 18 | 22 | Ø Ø1Ø Ø1Ø | 53 | 65 | Ø 11Ø 1Ø1 |
| 19 | 23 | Ø Ø1Ø Ø11 | 54 | 66 | Ø 11Ø 11Ø |
| 2Ø | 24 | Ø Ø1Ø 1ØØ | 55 | 67 | Ø 11Ø 111 |
| 21 | 25 | Ø Ø1Ø 1Ø1 | 56 | 7Ø | Ø 111 ØØØ |
| 22 | 26 | Ø Ø1Ø 11Ø | 57 | 71 | Ø 111 ØØ1 |
| 23 | 27 | Ø Ø1Ø 111 | 58 | 72 | Ø 111 Ø1Ø |
| 24 | 3Ø | Ø Ø11 ØØØ | 59 | 73 | Ø 111 Ø11 |
| 25 | 31 | Ø Ø11 ØØ1 | 6Ø | 74 | Ø 111 1ØØ |
| 26 | 32 | Ø Ø11 Ø1Ø | 61 | 75 | Ø 111 1Ø1 |
| 27 | 33 | Ø Ø11 Ø11 | 62 | 76 | Ø 111 11Ø |
| 28 | 34 | Ø Ø11 1ØØ | 63 | 77 | Ø 111 111 |
| 29 | 35 | Ø Ø11 1Ø1 | 64 | 1ØØ | 1 ØØØ ØØØ |
| 3Ø | 36 | Ø Ø11 11Ø | 65 | 1Ø1 | 1 ØØØ ØØ1 |
| 31 | 37 | Ø Ø11 111 | 66 | 1Ø2 | 1 ØØØ Ø1Ø |
| 32 | 4Ø | Ø 1ØØ ØØØ | 67 | 1Ø3 | 1 ØØØ Ø11 |
| 33 | 41 | Ø 1ØØ ØØ1 | 68 | 1Ø4 | 1 ØØØ 1ØØ |
| 34 | 42 | Ø 1ØØ Ø1Ø | 69 | 1Ø5 | 1 ØØØ 1Ø1 |

## Appendix B.

### Powers of 2

| n | $2^n$ | | n | $2^n$ | |
|---|---|---|---|---|---|
| 0 | 1 | $=8^0$ | 30 | 1 073 741 824 | $=8^{10}$ |
| 1 | 2 | | 31 | 2 147 483 648 | |
| 2 | 4 | | 32 | 4 294 967 296 | |
| 3 | 8 | $=8^1$ | 33 | 8 589 934 592 | $=8^{11}$ |
| 4 | 16 | | 34 | 17 179 869 184 | |
| 5 | 32 | | 35 | 34 359 738 368 | |
| 6 | 64 | $=8^2$ | 36 | 68 719 476 736 | $=8^{12}$ |
| 7 | 128 | | 37 | 137 438 953 472 | |
| 8 | 256 | | 38 | 274 877 906 944 | |
| 9 | 512 | $=8^3$ | 39 | 549 755 813 888 | $=8^{13}$ |
| 10 | 1 024 | | 40 | 1 099 511 627 776 | |
| 11 | 2 048 | | 41 | 2 199 023 255 552 | |
| 12 | 4 096 | $=8^4$ | 42 | 4 398 046 511 104 | $=8^{14}$ |
| 13 | 8 192 | | 43 | 8 796 093 022 208 | |
| 14 | 16 384 | | 44 | 17 592 186 044 416 | |
| 15 | 32 768 | $=8^5$ | 45 | 35 184 372 088 832 | $=8^{15}$ |
| 16 | 65 536 | | 46 | 70 368 744 177 664 | |
| 17 | 131 072 | | 47 | 140 737 488 355 328 | |
| 18 | 262 144 | $=8^6$ | 48 | 281 474 976 710 656 | $=8^{16}$ |
| 19 | 524 288 | | 49 | 562 949 953 421 312 | |
| 20 | 1 048 576 | | 50 | 1 125 899 906 842 624 | |
| 21 | 2 097 152 | $=8^7$ | 51 | 2 251 799 813 685 248 | $=8^{17}$ |
| 22 | 4 194 304 | | 52 | 4 503 599 627 370 496 | |
| 23 | 8 388 608 | | 53 | 9 007 199 254 740 992 | |
| 24 | 16 777 216 | $=8^8$ | 54 | 18 014 398 509 481 984 | $=8^{18}$ |
| 25 | 33 554 432 | | 55 | 36 028 797 018 963 968 | |
| 26 | 67 108 864 | | 56 | 72 057 594 037 927 936 | |
| 27 | 134 217 728 | $=8^9$ | 57 | 144 115 188 075 855 872 | $=8^{19}$ |
| 28 | 268 435 456 | | 58 | 288 230 376 151 711 744 | |
| 29 | 536 870 912 | | 59 | 576 460 752 303 423 488 | |

Given the byte size = n: range for unsigned integers is 0 to $2^n-1$
range for signed integers is $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$

Given the maximum absolute value such that $2^{n-1} \leq |\text{maximum}| < 2^n$:
byte size for unsigned integers is n
byte size for signed integers is n+1

## Appendix C.

## Internal Keycodes

If you wish to examine and interpret the contents of a variable, some octal representations of internal keycodes are given here for convenience. Complete tables are given in "aids." Use of these codes should be limited to inspection of variables. In -search-, for example, use "search '=',..." rather than "search o54ØØØØØØØØØØØØØØØØØØ,..." and "search 'cat',..." rather than "search oØ3Ø124ØØØØØØØØØØØØØØ,..." Not only is the tag easier to read and interpret, but problems arising from possible future changes in the internal codes will be avoided.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| shift | o7Ø | a | oØ1 | A | o7ØØ1 | Ø | o33 | ( | o51 |
| micro | o76 | b | oØ2 | B | o7ØØ2 | 1 | o34 | ) | o52 |
| font | o75 | c | oØ3 | C | o7ØØ3 | 2 | o35 | [ | o61 |
| superscript | o67 | d | oØ4 | D | o7ØØ4 | 3 | o36 | ] | o62 |
| subscript | o66 | e | oØ5 | E | o7ØØ5 | 4 | o37 | { | o7661 |
| locked superscript | o7Ø67 | f | oØ6 | F | o7ØØ6 | 5 | o4Ø | } | o7662 |
| locked subscript | o7Ø66 | g | oØ7 | G | o7ØØ7 | 6 | o41 | π | o762Ø |
| space | o55 | h | o1Ø | H | o7Ø1Ø | 7 | o42 | ° | o7617 |
| backspace | o74 | i | o11 | I | o7Ø11 | 8 | o43 | \| | o767Ø11 |
| halfspace | o7655 | j | o12 | J | o7Ø12 | 9 | o44 | ≡ | o7652 |
| half backspace | o7674 | k | o13 | K | o7Ø13 | + | o45 | ← | o767ØØ1 |
| | | l | o14 | L | o7Ø14 | - | o46 | → | o767ØØ4 |
| | | m | o15 | M | o7Ø15 | * | o47 | ~ | o7677 |
| | | n | o16 | N | o7Ø16 | × | o64 | % | o63 |
| | | o | o17 | O | o7Ø17 | / | o5Ø | $ | o53 |
| | | p | o2Ø | P | o7Ø2Ø | ÷ | o6Ø | ⇦ | o65 |
| | | q | o21 | Q | o7Ø21 | = | o54 | . | o57 |
| | | r | o22 | R | o7Ø22 | ≠ | o7654 | , | o56 |
| | | s | o23 | S | o7Ø23 | < | o72 | ? | o7Ø5Ø |
| | | t | o24 | T | o7Ø24 | > | o73 | ! | o7Ø57 |
| | | u | o25 | U | o7Ø25 | ≤ | o7672 | ; | o77 |
| | | v | o26 | V | o7Ø26 | ≥ | o7673 | : | o7Ø77 |
| | | w | o27 | W | o7Ø27 | | | ' | o7Ø42 |
| | | x | o3Ø | X | o7Ø3Ø | | | " | o7Ø56 |
| | | y | o31 | Y | o7Ø31 | | | [illegible] | o7Ø41 |
| | | z | o32 | Z | o7Ø32 | | | | |