AutoCAD and binaries

By Stig Madsen, 2001
(Rev. december 2002)

This document will try to shed some light on binaries, bitcoded values in AutoCAD and the functions provided by AutoLISP: logand, logior, lsh and boole.
If you are not familiar with the binary system and conversions between binary and decimal, it is recommended that you read through the following section. If you don't care for lengthy explanations or are well at home with binaries, you can skip to the section "Logical operators", and, if that doesn't capture your interest either, then at least read the section "Masking integers with logical operators".

Decimals and binaries

In the decimal system we have 10 symbols to juggle with. We start counting from 0 to 9 and when we run out of symbols we simply shift to the left and start all over again to the right increasing the left number whenever we run out of symbols to the right. When the left number runs out of symbols too we simply shift one more time to the left and so on. Fortunately this shifting can be represented mathematically by using the power of a base number, in this case the number that presents itself when we have shifted one time, the number 10. The number of times the array of symbols (0-9) has been shifted to the left indicates the power in which to raise the number 10. When no shifting has occurred we are in the place of 10 raised to 0 = 1, also known as the one's place. When shifted once we are in the place of 10 raised to 1, - known as the ten's place. Next shifting results in 10 raised to 2, which is the hundred's place and so on. Confusing? Not really because this is the way we were taught to do number crunching from the time the doctor said, oh look, he came out all right with 10 fingers!

Counting numbers in a decimal system can be converted into "power of base number" notation like this:

```
0     =     10^0*0                                          =    0
1     =     10^0*1                                          =    1
....
9     =     10^0*9                          =    9
10    =     10^0*0 + 10^1*1                 =    0+10       =    10
11    =     10^0*1 + 10^1*1                 =    1+10       =    11
....
20    =     10^0*0 + 10^1*2                 =    0+20       =    20
...
99    =     10^0*9 + 10^1*9                 =    9+90       =    99
100   =     10^0*0 + 10^1*0 + 10^2*1        =    0+0+100    =   100
101   =     10^0*1 + 10^1*0 + 10^2*1        =    1+0+100    =   101
etc. etc.
```

It all changed sometime in the late 40's when someone began to mass-produce a number-crunching machine using only 2 symbols, on and off - or as we like to call it 1 and 0, thereby introducing the binary system to the masses. The same mathematics controlling the decimal system also applies to the binary system. When we run out of available symbols we simply shift to the left until running out again, shifting once more and so on. Only difference is that the binary system uses only 2 symbols, 0 and 1. This means that when shifting one place we have only counted 2 times thus making our base number 2. So, one shift is 2 raised to 1, two shifts are 2 raised to 2, three shifts are 2 raised to 3 and so on. Illustrating the procedure of counting in a binary system looks like this, starting with 0, and converted into "power of base number" notation (which, fortunately for computers of flesh and blood, can be used to convert directly into the decimal system!) :

```
0     =   2^0*0                                          =   0
1     =   2^0*1                                          =   1
10    =   2^0*0 + 2^1*1                  =   0+2         =   2
11    =   2^0*1 + 2^1*1                  =   1+2         =   3
100   =   2^0*0 + 2^1*0 + 2^2*1          =   0+0+4       =   4
101   =   2^0*1 + 2^1*0 + 2^2*1          =   1+0+4       =   5
110   =   2^0*0 + 2^1*1 + 2^2*1          =   0+2+4       =   6
111   =   2^0*1 + 2^1*1 + 2^2*1          =   1+2+4       =   7
1000  =   2^0*1 + 2^1*0 + 2^2*0 + 2^3*1  =   0+0+0+8     =   8
1001  =   2^0*1 + 2^1*0 + 2^2*0 + 2^3*1  =   1+0+0+8     =   9
etc. etc.
```

To convert from decimal to binary is a bit more cumbersome affair than the binary-decimal conversion and it will be left out here as it isn't all that needed for the purpose of looking at AutoLISP's logical functions. Many fine books, papers and websites describe the procedure.

As computer parts only know how to deal with power off and power on they use these states for all the tasks they are capable of doing. Electrical currents running through the circuits of a computer can be alternated and changed in various ways. The methods of calculating which turns electrical currents shall take within various components are known as logical operations. By that it becomes possible to apply mathematics to how electrical currents can be redirected within our machines. The kind of mathematics performed this way is often referred to as Boolean logic or Boolean algebra.

Each digit of a binary number is in computer-related terms called a bit. As a bit is really a switch describing the presence or absence of an electrical current it is practical to assemble these switches (bits) into chunks, which can then be processed as a unit and, much like a genome, can represent a unique transaction within the processing unit. Naturally this comes in handy for the human on the other side of the output device because once processed, chunks of these bits can also represent a color on the screen, an entity type in AutoCAD, a pen number, a character or a number that humans can use.
Bits are assembled into chunks the size of 2^3 bits = 8 bits, also called one byte (also often called one character). With eight bits a range of 2^8 = 256 different values can be achieved. To describe even more values bytes are joined together to form different data types - integers ranging 2^16 values (two byte integers) and 2^32 values (surprisingly, four byte integers), floating decimal numbers with 4 or more bytes etc. etc.
In any case, how long a data type might be, binary numbers are always forming an array of 1's and 0's on which logical operations can be performed.

Logical operators

Some of the most important logical operators for the purpose of digging into AutoLISP's binary handling are AND, OR, XOR, NOR and NOT.

The first four operators take as input two bits and return one resulting bit. NOT processes a single bit and returns the "opposite value" of the bit.
Note: Don't confuse AutoLISP's NOT function with logical NOT. AutoLISP’s NOT function merely inspects the value of an item and returns T if it amounts to nil. In AutoCAD the logical NOT function is ~ .

If the value 1 is considered true and 0 is considered false then a textual description of the operators would be like this:

AND: returns true only when both input bits are true
OR: returns true if both or just one of the input bits are true
XOR: returns true if only one of the input bits is true
NOR: returns true only when both input bits are false
NOT: returns true if the input bit is false, otherwise it returns true

OR is known as inclusive OR (hence the "i" in AutoLISP's function LOGIOR), and XOR is known as exclusive OR.

Applied to what is commonly know as a truth table, the four operators return the values shown below. The two rightmost columns are input bits and the last column is the output bit. Of course, in the case of NOT there is only one input bit.

```
AND:                    OR:
0     0  =  0            0     0  =  0
0     1  =  0            0     1  =  1
1     0  =  0            1     0  =  1
1     1  =  1            1     1  =  1

XOR:                    NOR:                    NOT:
0     0  =  0            0     0  =  1            0  =  1
0     1  =  1            0     1  =  0            1  =  0
1     0  =  1            1     0  =  0
1     1  =  0            1     1  =  0
```

It is important to point out that these truth tables only process a single bit. While bits are actually processed one by one in the innermost parts of a computer, it is wrong to assume that the same values will appear when using the AutoLISP logical

functions. In AutoLISP the NOT operator is in the form of the function `(~ int)`. As seen it takes only one argument and, more importantly, the argument is an integer – not a single bit. Fire up your AutoCAD and try it out with a value of 0. According to the truth table it should return 1, but darn, it returns –1! What's happening?

```
(~ 0)
-1
```

As the AutoLISP Help Reference points out, the argument is an integer and earlier we learned that integers consist of more than one bit. But how can you really talk about true and false in relation to integers? Is 48 true, and is it more true than 13? The integer is a specific data type with a fixed number of bits. In AutoLISP integers are 32-bit signed numbers, so no matter if all 32 bits in an integer are used it will always consist of 32 bits. Consequently the number 0 is also made up of 32 bits (which all have the value of 0). When you apply a logical operator to an integer, it will run through every single bit and process it without changing the positions of the bits. The return value will be an integer with the same amount of bits, but with the bits changed according to the logical operation performed on each bit. In the case of using the NOT operator on the value 0, here is what happens:

```
(~ 0)        NOT  00000000000000000000000000000000
-1             =  11111111111111111111111111111111
```

and when supplying the value 1 as an argument:

```
(~ 1)        NOT  00000000000000000000000000000001
-2             =  11111111111111111111111111111110
```

NOT performs on every single bit reversing their values but preserving their place in the array.

But why is the result negative? Because integers can describe both positive and negative values, the computer has to know how to tell the difference when looking at an integer. Negative numbers are defined by setting the leftmost bit to 1 – this bit is called the sign bit. Popular speaking, whenever the computer is given an integer it looks at the sign bit to determine whether the value is positive (sign bit is 0) or negative (sign bit is 1). Without going deeper into how this works and why negative numbers aren't just positive numbers with a sign bit, the computer uses a technique called two's complement allowing it to do arithmetic with both positive and negative values.

For our particular use the most important thing to keep in mind is that AutoLISP's logical operators only work with integers. Although it may seem that way, this text is not intended to give an in-depth knowledge of how binaries tumble around in a computer, but solely to give enough background information on how to extract bitcoded values in AutoLISP and AutoCAD!

Masking integers with logical operators

As seen, logical operators can be used to manipulate integers at bit levels, either investigating or changing any given bit to suit any need.

Note: While integers are 32 bit long, in the following text every array of bits is representing an integer. For clarity, and for the sake of my sanity while typing, they are truncated into a single byte. This means that you should pretend the following integers to be preceded by 24 zeros, so if you see a 1 in the leftmost place it is *not* a negative value unless otherwise is specifically stated. The 24 leftmost bits are just left out.

Suppose we have a byte (read: an integer) in which to figure out if a specific bit is set or not. How would we do this? Simple: the AND operator returns 1 if and only if both input bits are 1. For instance, find out if the 2nd bit of the number 182 is set (2nd bit = 1)?

182 in binary notation is 10110110. When we do an AND operation with this number and a number where only the 2nd bit is set then we will end up either with the second number itself (in case the 2nd bit is set) or with 0 (if the 2nd bit is not set):

```
10110110 AND          decimal:  182 AND
00000100 =                        4 =
--------                        ---
00000100                          4
```

If any 1's in the first byte corresponds to 1's in the second byte, they are “transferred” into the return value while forcing all other bits to 0's. This technique is known as masking. With the AND operation we simply apply an integer with only those bits set that we want to investigate in the first integer.
Because only one bit was investigated and the resulting number is the same as the number used to mask out the binary representation of 182 (10110110), we know that the second bit is set. By the way, for the purpose of this text, bits are indexed from right to left starting with bit 0. In a byte the leftmost bit is therefore bit 7.
The AutoLISP function LOGAND does the logical AND operation for us:

```
(logand 182 4)
4
```

In addition to the two required inputs, LOGAND accepts numerous inputs and does an AND operation with all. Given the truth table more inputs than 2 will still result in true if and only if all bits in the same place are 1, example:

```
(logand 46 182 38 170)
34
```

In binary notation:

```
00101110 AND          decimal:   46 AND
10110110 AND                    182 AND
00100110 AND                     38 AND
10101010 =                      170 =
--------                        ---
00100010                         34
```

Similarly, we can set a specific bit in a number by using the OR operator with a mask containing the specific bit with a value of 1. Again, using 182 but this time setting the 3rd bit (decimal value 8) in the returned integer:

```
10110110 OR           decimal:  182 OR
00001000 =                        8 =
--------                        ---
10111110                        190
```

Corresponding bits that are both 0 are left alone and any other bits return 1 if either or both bits are 1. In AutoLISP the LOGIOR function performs the inclusive OR operation:

```
(logior 182 8)
190
```

Like LOGAND it can operate with numerous values. Again, according to the truth table it will return 1 if only one of all the bits in the same place is set:

```
(logior 46 182 38 170)
190
```

In binary notation:

```
00101110 OR           decimal:   46 OR
10110110 OR                     182 OR
00100110 OR                      38 OR
10101010 =                      170 =
--------                        ---
10111110                        190
```

As stated at the beginning of this document, AutoLISP provides more logical functions than the 3 seen above and we’ll run through them in the following as we stumble upon them.

Logical operators and AutoCAD

Now, you may think: What a lot of rubbish – good thing I didn't waste time reading all that! Well, go back and study it, because here is how you can use all these bits and pieces in your code.

Coming across some of the flags used in AutoCAD entities and table entries as described in the DXF References, you will notice that some of them (e.g. in many cases code 70) are stated to be bit coded, but the flags are listed as decimal integers. How can that be? This is my best bid: Instead of supplying raw bit-handling functions which cannot be used anywhere else in AutoLISP, it offers to manipulate the bits using integers and the bit-handling functions themselves return integers. Because of this you don't have to convert any decimal numbers into binary or hexadecimal numbers and convert them back again.
This is the reason that the logical operation samples listed above are also shown with decimal values.

If you have an entity in AutoCAD and want to check it for a particular bit coded flag all you have to do is to check if the flag is "contained" within the value supplied by its entity list structure.
Suppose a bit coded value for an entity has the value of 84, which bits are set? First we need the appropriate operator. Above we learned that AND can do this for us, that is if A AND B = B then A and B has the same bit set. All we have to do is to make sure that B only has the bit set that we are looking for.

Because we are talking about bit coded values in AutoCAD we want it to return a list of the possible flags in decimal: 1, 2, 4, 8, 16, 32, 64 etc. etc. Here's a shot at making a lisp routine to check all bits in a particular value:

```
(defun findBits (num / i a bitList)
  (setq i 0
        a 1
  )
  (while (>= num a)
    (if (= (logand num a) a)
      (setq bitList (cons a bitList))
    )
    (setq a (expt 2 (setq i (1+ i))))
  )
  (reverse bitList)
)
```

Explanation:
Variable `a` will be our masking number shifting 1 bit each time running through the loop (remember, shifting 1 bit means increasing the binary number with 2 raised to the power of x, where x is the position of a specific bit (hmmm, maybe you shouldn't have skipped the first part of this text?). The masking number will take on the value of all the possible flags as described above while testing a different flag in every loop.
Inside the loop it performs the logical AND operation on the value (`num`) and the mask (`a`) determining if the bit in `a` is also set in `num` . If this is true then the particular flag-value is added to a list.
All you have to do to check if a particular flag is present is to compare the flag-value to the list. Say, check for the flag-value 32 "in" the number 182:

```
(setq theFlags (findBits 182))
(2 4 16 32 128)
```

Perform the search for flag 32 in the returned list:

```
(if (member 32 theFlags)(princ "Flag 32 found")(princ "Bummer!"))
```

If you read the first part of this text you will know that shifting a bit in a binary number is the same as doubling the number each time a shift has occurred. AutoLISP provides a function, LSH, which shifts a bit either to the left (=increasing) or to the right (=decreasing). LSH stands for LeftSHift, but if the second argument is negative it will rightshift. The result of calling LSH numerous times starting with 1 and shifting 1 place each time looks like this in binary and decimal notation (preceding zero's are shown for clarity in binary notation):

```
(setq x 1)
(setq x (lsh x 1))   ; set x to x shifted left 1 time

number
of times    binary       decimal
0           00000001           1
1           00000010           2
2           00000100           4
3           00001000           8
4           00010000          16
5           00100000          32
etc...

(lsh 1 5) ; means leftshift the value 1 five times = 32 (as in the sequence above)
```

Notice how the bit 1 shifts place doubling the number each time it is shifted? We can use this in the findBits routine and toss away a variable:

```
(defun findBits (num / a bitList)
  (setq a 1)
  (while (>= num a)
    (if (= (logand num a) a)
      (setq bitList (cons a bitList))
    )
    (setq a (lsh a 1))  ; increase a with 2^(number of loops)
  )
  bitList
)
```

... and at the same time we learned something about LSH. AutoLISP's LSH operates on 32-bit integers and will continue shifting until all 32 bits have shifted. After 31 leftshifts the integer is shifted in again and on the 32nd shift the integer will reemerge – if rightshifting the bits will fall off the right edge and be lost:

```
(lsh 190 32)  ; returns 190, i.e. the same integer as we started out with
190

(lsh 190 57)  ; results in the same number as (lsh 190 (- 57 32))
2080374784
```

LSH uses two's complement, which means that if the rightmost bit in the return value is 1 it is regarded as being negative. In the case above where the number 1 is shifted to the left this happens when the bit has reached the leftmost place in a 32-bit integer (after shifting 31 times). Although (lsh x n) is the same as (* x (expt 2 n)), and (lsh x –n) is equivalent to (fix (/ x (expt 2 n))) be careful when shifting to negative values occur and only use LSH in well-chosen places.

Anyway, enough about shifting one way or another. We have already seen how to investigate a bit coded value, let's sum it up with a few useful applications of LOGAND and LOGIOR:

To check if a specific flag is set in a bitcoded integer use the above function or just write a conditional function like this one:

```
(defun checkFlag (num flag)
  (= (logand num flag) flag)
)
```

This functions will return T if `flag` is set in `num`, otherwise it returns nil. We have already seen during the first encounter of LOGAND that if we are masking a value with only one bit set in the masking value and the returned value comes out as our masking value then we know that the masked bit is set in the value:

```
10110110 AND         decimal:  182 AND
00000100 =                       4 =
--------                       ---
00000100                         4
```

There ya go, 3rd bit in the number 182 is set because AND returned the masking value. Applied to our newly written function it will look like this:

```
(checkFlag 182 4)
T
```

To check a group code 70 value in e.g. a polyline we could use this sequence:

```
...
  (setq entlist (entget (car (entsel "Select a polyline: ")))) ; get the polyline
  ;; code to check for entity type would go here...
  (setq num (cdr (assoc 70 entlist)))                          ; extract code 70 value
  (cond
    ((checkFlag num 1)(princ "Polyline is closed"))            ; check the flags
    ((checkFlag num 2)(princ "Polyline is curve-fitted))
    ;; ... more options go here
    ((checkFlag num 64)(princ "Entity is a polyface mesh"))
    (T nil)
  )
...
```

A note on this procedure, though: COND will stop after finding the first flag contained in the value. To test for more flagvalues don't use COND!

To set a specific flag in a bit coded integer we can use logical OR to mask the bits we want to set. Given OR's truth table LOGIOR will not affect bits that are already set:

```
(defun setFlag (num flag)
  (logior num flag)
)
```

This function will return `num` with the bits specified in `flag` turned on, i.e. changed to 1's. For example, to close a 3D mesh in the M direction set its group code 70 flag to 1 (supposing that only flag 16 is set indicating the entity is in fact a 3D mesh):

```
(setFlag num 1)
17
```

LOGIOR will change all the bits where one or both are 1. This is what happens at bit-level:

```
00010000 OR           decimal:  16 OR
00000001                         1 =
00010001                        17
```

Suppose we want to close the mesh in both M and N direction (flag 1 and 32) and the mesh that we picked is already closed in the N direction (flag 32 is already set):

```
(setFlag num (+ 1 32))
49
```

The following is what happens at bit-level. Notice that the 5th bit (flag 32) is already set in `num` and LOGIOR leaves it unchanged because 1 OR 1 = 1:

```
00110000 OR           decimal:  48 OR
00100001                        33 =
00110001                        49
```

Now we can use the following sequence to close a 3D mesh in both M and N direction:

```
...
  (setq entlist (entget (car (entsel "Select a 3D mesh: "))))
  ;; code to check for entity type would go here...
  ;; now check if we have a 3D mesh (code 70 should contain the flag 16)
```

```
  ;; using the checkFlag function from before:
  (setq num (cdr (assoc 70 entlist)))
  (if (checkFlag num 16)
     (setq ent (entmod (subst (cons 70 (setFlag num (+ 1 32)))
                              (assoc 70 entlist)
                      entlist)
               )
     )
     (princ "Selected entity is not a 3D mesh") ; if no flag 16
   )
...
```

Note: Always check ENTMOD to see if the operation failed and take the proper action in your code to ensure that ENTMOD does its job. When altering group code 70 flags like this have in mind that "incompatible" flag values are common. As an example, a polyline can't both be a 3D mesh and a polyface - therefore the flags 16 and 64 won't show up in a polyline at the same time. Attempts on setting a group code 70 to 16+64 on a polyline with the code above ENTMOD will not produce a nil return value, but AutoCAD will not set the flags either, so don't assume that the procedure succeeded. Instead check for success or failure in some other ways; checking the group code after attempting to change it would be an obvious method.

And now to something completely similar: BOOLE

BOOLE is AutoLISP's general function for logical bitwise operations. It can handle 16 Boolean operations for which only a few I will live to tell about, but it is fairly easy to set up truth tables if you want to explore all the functions. Otherwise it will be necessary to refer to more specialized literature.

All functions in BOOLE are input as an integer argument. If you don't have the AutoLISP Reference at hand the syntax for BOOLE is as follows:

```
(boole operator int1 [int2 ... ])  ; my apologies to the late George Boole for not using capital B
```

The 4 most common logical operators AND, XOR, OR or NOR has the following integer representations. Please refer to the text and the truth tables above for an explanation of these operators.

AND	1
XOR	6
OR	7
NOR	8

(boole 1 ...) and (boole 7 ...) performs the same operations as LOGAND and LOGIOR. As mentioned before, beware that for example NOR does a one's complement on a 16 bit integer and the truth table when printed out with BOOLE is not the same as making a truth table on a single bit. We can illustrate this is if doing a NOR operation on a byte – and by the way, A NOR B is the same as NOT(A OR B):

```
A   00000001  NOR       decimal: 1
B   00000000  =>                  0
A   00000001  OR                  1
B   00000000  =                   0
    00000001  NOT                 1
=   11111110                     -1
```

Remember, binary representation of a negative integer is (in popular terms) defined by setting the leftmost bit to 1, so AutoLISP will regard the result of this equation as being two's complement of 1 = –1.

To try and make a truth table lisp showing all different values of the different BOOLE operators, you could make a defun like the following. If you would explore this matter even further it is possible to write a decimal-to-binary conversion routine and really check out the binary values bit for bit (hint: use `(expt 2 n)` to run through all bits).

```
(defun nothingButTheTruth (/ a)
```

```
  (defun princline (b n m / x)
    (setq x (boole b n m))
    (cond ((= x -1)(setq x 1))
          ((= x -2)(setq x 0)))
    (princ (strcat "\n" (itoa n) "    " (itoa m) "    " (itoa x)))
  )

  (setq a 0)
  (repeat 16
    (princ (strcat "\nOperator: " (itoa a)))
    (princline a 0 0)
    (princline a 0 1)
    (princline a 1 0)
    (princline a 1 1)
    (terpri)
    (setq a (1+ a))
  )
  (princ)
)
```

It will produce an output like this:

```
Operator: 0
0    0    0
0    1    0
1    0    0
1    1    0

Operator: 1
0    0    0
0    1    0
1    0    0
1    1    1

Operator: 2
0    0    0
0    1    0
1    0    1
1    1    0
...etc...

Operator: 8
0    0    1
0    1    0
1    0    0
1    1    0
...etc...
```

... and so on. A full list of the operators can be found at the end of this article.

Whether you use BOOLE operators 1 and 7 or you use LOGAND and LOGIOR is indifferent. Of course, in some specific code where it is needed to offer the operator as a variable only BOOLE will do the job.

Clearing bits in a number can be done in a number of ways. For example, clearing the 4th bit with AND in the number 48 could be done like this:

```
(logand 48 -17)
32
```

Why exactly use -17 as the masking value?, Well, looking at it bit-wise this is what takes place:

```
00110000 AND          decimal:  48 AND
11101111                       -17 =    (have in mind that integers are 32 bit and not 8 bit long)
00100000                        32
```

We simply take advantage of the fact that AND clears all bits if they aren't both 1's. Of course, this can be done in a more obvious and especially a much more useful way when using logical NOT to inverse the flag we want to mask with:

```
(logand (48 (~ 16)))
32

00010000 NOT          decimal:  16 NOT
11101111 AND                   -17 AND (again, have in mind that integers are 32 bit and not 8 bit long)
00110000 =                      48 =
00100000                        32
```

Does this seem confusing? You betcha. Why not take advantage of the facts that LOGAND returns 0 if the bit value we're searching for is not present in the integer, and it returns the bit value itself if it is present in the integer? This means we can simply subtract LOGAND's return value from the integer itself:

```
(- 48 (logand 48 16))
32
```

For bits that are not set in the integer, LOGAND will return 0 which doesn't harm the output. For example, the 4th bit in the integer 47 is not set (in 47 bits 0, 1, 2, 3 and 5 are all set, but not bit 4), thus LOGAND will return 0 in this case:

```
(- 47 (logand 47 16))
47
```

So, clearing a bit with LOGAND is easily done by subtracting `(logand num bit)` from `num`. If clearing more than 1 bit from an integer, we can simply add together the bit values. For example, to clear the bits 0, 2 and 3 (i.e. the integer bitvalues 1, 4 and 8) from 47:

```
(- 47 (logand 47 (+ 1 4 8)))
34
```

Now we can define a simple function that clears a bit from an integer:

```
(defun clearBit (num flag)
  (- num (logand num flag))
)
```

If we look at the truth tables returned by BOOLE in the function `nothingButTheTruth` above, it becomes obvious that BOOLE's operator number 2 also can do the job for us. It returns true if and only if the first bit is 1 and the second bit is 0:

```
(defun clearBit (num flag)
  (boole 2 num flag)
)
```

This function will perform exactly the way as the previous `clearBit` function with LOGAND. It returns `num` with the flag value or values cleared indicated by argument `flag`. For instance:

```
(clearBit 48 16)
32

00110000 BOOLE 2      decimal:  48 BOOLE 2
00010000                        16 =
00100000                        32
```

and

```
(clearbit 47 (+ 1 4 8))
34
```

For both functions, it is important that the order of arguments be obeyed. `(clearBit 16 48)` is not the same as `(clearBit 48 16)`. This may seem like a trivial point, but when dealing with LOGAND and LOGIOR where order of arguments is as indifferent as in arithmetic addition, it should be mentioned.

By the way, to clear all bits you can use the maximum integer 2147483647, or as returned by the expression `(~ (expt 2 31))`. This is only superficial information, though, because it is much easier just to set the value to 0!

BOOLE operators

Not that it comes up often when programming in AutoLISP (far from it), but when wanting a particular logical operation besides NOT (logical), LOGAND and LOGIOR it will sometimes be necessary to use BOOLE. Either in one operation or as nested operations. In any case it would be much desired if the various operators could be documented in a more precise way than Autodesk has done it. So here's a list of all 16 operators!
Where the operation can be expressed by logical formulas, these formulas are shown in LISP style format but with standard operator names. These names are NOT to be confused with AutoLISP functions. Where order is essential, the first input bit is always called A and the second is called B as shown in the truth tables.

Operator 0: NULL

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	0

NULL operator. Also called CLEAR as it clears all inputs no matter the input bits.

Operator 1: AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

This operator is the same as the AutoLISP function LOGAND.

Operator 2: AND-2 (in Common LISP called BOOLE-ANDC2)

A	B	Out
0	0	0
0	1	0
1	0	1
1	1	0

(AND A (NOT B))

Operator 3: Only A (in Common LISP called BOOLE-1)

A	B	Out
0	0	0
0	1	0
1	0	1
1	1	1

(OR (AND A B) (AND A (NOT B))) = A. This operator returns only first input bit.

Operator 4: AND-1 (in Common LISP called BOOLE-ANDC1)

A	B	Out
0	0	0
0	1	1

1	0	0
1	1	0

(AND (NOT A) B)

Operator 5: Only B (in Common LISP called BOOLE-2)

A	B	Out
0	0	0
0	1	1
1	0	0
1	1	1

(OR (AND A B) (AND (NOT A) B)) = B. This operator returns only second input bit.

Operator 6: XOR (also called Exclusive OR)

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive OR can be constructed with this equation: (OR (AND A (NOT B))(AND (NOT A) B))

Operator 7: OR (also called Inclusive OR)

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

(OR A B). This operator is the same as the AutoLISP function LOGIOR.

Operator 8: NOR (also sometimes referred to as NOT-OR)

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

(NOT (OR A B))

Operator 9: XNOR (also called Exclusive NOR)

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	1

(OR (AND A B) (AND (NOT A)(NOT B)))

Operator 10: NOT B (also known as Inverting buffer for second bit)

A	B	Out
0	0	1

0	1	0
1	0	1
1	1	0

(NOT B). This operator discards first input put and only returns the inverted second input bit. In Common LISP it is called BOOLE-C2.

Operator 11: ? (in Common LISP called BOOLE-ORC2)

A	B	Out
0	0	1
0	1	0
1	0	1
1	1	1

(OR A (NOT B))

Operator 12: NOT A (also known as Inverting buffer for first bit)

A	B	Out
0	0	1
0	1	1
1	0	0
1	1	0

(NOT A). This operator discards second input put and only returns the inverted first input bit. In Common LISP it is called BOOLE-C1.

Operator 13: ? (in Common LISP called BOOLE-ORC1)

A	B	Out
0	0	1
0	1	0
1	0	1
1	1	1

(OR (NOT A) B)

Operator 14: NAND (sometimes referred to as NOT-AND)

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

(NOT (AND A B))

Operator 15: SET (also known as Identity)

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	1

This operator has no formula derived from other logical operations – it merely disregards any input bits and returns a set bit.

To wrap it up...

With this document it's my hope that the use of bit coded values in AutoCAD and to investigate, extract and alter them with AutoLISP is somewhat covered and, more importantly, explained adequately to make use of the topics in a practical way.

If you still want to explore and manipulate for example group code 70 values but you absolutely refuse to use logical operators then you could begin to explore the ActiveX functionality, either in VLISP or VBA. Using ActiveX, the properties of an entity are given by named constants, but that is the only thing said on that matter in this document.
