PharmaSUG China 2019 - Paper AD-61

A Practical Use of Bitwise Encoding in SAS®

Rowland Hale, Syneos Health, Berlin, Germany

ABSTRACT

Bitwise encoding provides a convenient way to encapsulate multiple flags within a single numeric code. A good example of bitwise encoding in action is SYSINFO, the automatic macro variable which is returned by the COMPARE procedure in SAS®. The code it contains is a number whose bits have been set to encode flags which signify the failure of specific aspects of the comparison such as differing data set labels or conflicting variable types. Another example is VBScript's MsgBox function which uses a similar approach to specify the buttons and icon for the dialog box it displays. After a brief look at how SYSINFO works, we move on to see another practical use of bitwise encoding where the technique is applied to facilitate the validation of option combinations contained within a macro variable. Benefits of the approach are ease of maintenance, readability and scalability should new options or option combinations be added in the future.

INTRODUCTION

This paper takes a look at bitwise encoding and uses two examples, one from SAS and one from elsewhere, to explain the bitwise encoding concept. The paper moves on to explain how the technique can be applied to a real-world use case, namely validating a macro parameter value where expected values are specific combinations of supported options. The paper is aimed at intermediate level SAS programmers. Knowledge of regular expressions would be helpful; although some explanation is given basic knowledge is assumed.

A QUICK LOOK AT BINARY REPRESENTATION

Of course we are all familiar with the decimal counting system:

	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
Decimal notation:	100000000	10000000	1000000	100000	10000	1000	100	10	1

Table 1. Column values in the decimal counting system

As we know, the decimal system uses the 10 numerals 0-9, each "column" in a decimal number representing 10 to the power of the column number minus 1. For a number of reasons however, computers utilise the binary counting system, which uses, as the name suggests, just the two numerals 0 and 1. This time the columns represent 2 to the power of the column number minus 1:

	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary notation:	100000000	10000000	1000000	100000	10000	1000	100	10	1
Decimal notation:	256	128	64	32	16	8	4	2	1

Table 2. Column values in the binary counting system

This happens to be very handy in electronics because each so-called "bit" (i.e. what we call a column in decimal counting) can be set to reflect either the ON state (1) or the OFF state (0). But we are not so much interested in electronics and how computers work at that level as we are in programming for real world data, so why is this relevant to us? Well, it only takes a little imagination to see that bits in a number can also be set so as to "flag" data which is more meaningful to us, namely real world data, and because we have quite a few of bits at our disposal in even quite small numbers, we can carry a fair amount of flagging information in that number. Doing this can be referred to as bitwise encoding, as we are encoding data into a number by setting each bit in that number to reflect the data, or flags, we wish to carry.

A GOOD EXAMPLE OF BITWISE ENCODING IN SAS

To find a good example of bitwise encoding in SAS we need look no further than the COMPARE procedure. As many readers will know, this procedure compares two data sets and reports the differences it finds in a number of ways:

- As a report to the OUTPUT window
- As a data set

The procedure also returns a value in the automatic macro variable SYSINFO. This macro variable contains a decimal integer which uses – you guessed it – bitwise encoding to return the various types of difference the procedure found between the two data sets it compared, each difference type resulting in a specific bit being set. SYSINFO can be used in scenarios involving a requirement to automate data set comparison where only difference types are to be reported. The following table describes the meaning of each bit:

Bit	Code	Binary	Description
1	1	0000000000000001	Data set labels differ
2	2	0000000000000010	Data set types differ
3	4	0000000000000100	Variable has different informat
4	8	0000000000001000	Variable has different format
5	16	0000000000010000	Variable has different length
6	32	0000000000100000	Variable has different label
7	64	0000000001000000	Base data set has observation not in comparison
8	128	0000000010000000	Comparison data set has observation not in base
9	256	0000000100000000	Base data set has BY group not in comparison
10	512	0000001000000000	Comparison data set has BY group not in base
11	1024	0000010000000000	Base data set has variable not in comparison
12	2048	0000100000000000	Comparison data set has variable not in base
13	4096	0001000000000000	A value comparison was unequal
14	8192	0010000000000000	Conflicting variable types
15	16384	0100000000000000	BY variables do not match
16	32768	1000000000000000	Fatal error: comparison not done

Table 3. The meaning of each bit in SYSINFO, the code returned by the COMPARE procedure in SAS

As we have already discovered, the beauty of bitwise encoding is that multiple bits can be set as required thus enabling multiple pieces of information to be carried. This means, in the case of SYSINFO, that the single return value contains information about all 16 difference types which are supported within the variable by virtue of each bit being set or not depending on the outcome of the comparison.

For example, if the COMPARE procedure determines that the compared data sets have the following differences:

- The data sets have different data set labels
- A common variable within both data sets has different lengths
- A value comparison revealed unequal values

bits will be set as follows:

Bit	Code	Binary	Description
1	1	0000000000000001	Data set labels differ
5	16	0000000000010000	Variable has different length
13	4096	0001000000000000	A value comparison was unequal
Return code:	**4113**	0001000000010001	Sum of the separate difference codes

Table 4. Example SYSINFO return code reflecting various difference types found during a comparison

Let's test this with a simple data set comparison:

```
data d1 (label=Base data set);
    greeting = 'Hello';
run;

data d2 (label=Comparison data set);
    greeting = 'Goodbye';
run;

proc compare base=d1 comp=d2 noprint;
run;

%put SYSINFO >>> &sysinfo;
```

Here's the log output:

```
1          proc compare base=d1 comp=d2 noprint;
2          run;

NOTE: There were 1 observations read from the data set WORK.D1.
NOTE: There were 1 observations read from the data set WORK.D2.
NOTE: PROCEDURE COMPARE used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

3
4          %put SYSINFO >>> &sysinfo;
SYSINFO >>> 4113
```

We will see how to interpret the SYSINFO return code in a later section.

ANOTHER EXAMPLE OF BITWISE ENCODING (NOT SAS)

Outside of SAS, we can see another example of the bitwise encoding principle in action in VBScript's MsgBox function:

Value	Action	Constant
Buttons:		
0	Show OK button	VBOKOnly
1	Show OK and cancel buttons	VBOKCancel
2	Show abort, retry, ignore buttons	VBAbortRetryIgnore
3	Show yes, no, cancel buttons	VBYesNoCancel
4	Show yes, no buttons	VBYesNo
5	Show retry, cancel buttons	VBRetryCancel
Icon:		
16	Show critical message icon	VBCritical
32	Show question message icon	VBQuestion
48	Show exclamation message icon	VBExclamation
64	Show information message icon	VBInformation
Button default:		
0	First button is default	VBDefaultButton1
256	Second button is default	VBDefaultButton2
512	Third button is default	VBDefaultButton3
768	Fourth button is default	VBDefaultButton4
Modality:		
0	Demands that the user respond to the dialog before allowing continuation of work in current application	VBApplicationModal
4096	Causes suspension of all applications until the user responds to the dialog	VBSystemModal

Table 5. Values used to define a message box generated by the MsgBox function in VBScript

Things aren't quite so straightforward here, as values within each category are mutually exclusive, but values across categories can be summed into a cumulative value, all possible combinations of which, as with SYSINFO, result in a unique value which carries various flags within it.

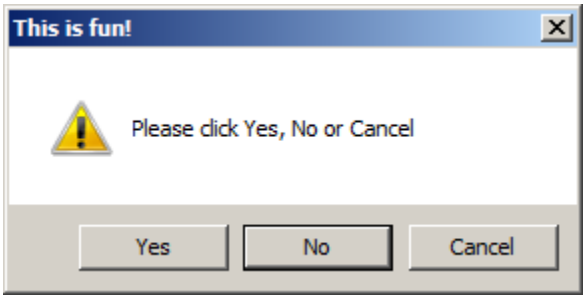
For example, the value needed to generate a message box with YES, NO and CANCEL buttons and an exclamation icon where the NO button is set as the default is as follows:

3 + 48 + 256 = 307

The following call to MsgBox:

```
MsgBox "Please click Yes, No or Cancel", 307, "This is fun!"
```

results in the following message box being displayed:



Display 1. The message box generated by the described MsgBox function

To try this yourself, copy the above function call into a text file with the extension `.vbs` and run the file.

HOW TO DECODE A BITWISE VALUE

Returning back to SYSINFO, once we've run PROC COMPARE how do we then interpret SYSINFO's value to establish which difference types the procedure found? To do this, we need to check each bit in the value to determine whether or not it has been set. There are several ways to do this:

Use bitmasks

In SAS, a bitmask, or bit string constant, is one of several types of SAS constant. It takes the form of a series of zeros, ones and dots in quotation marks followed immediately by an upper or lower case B where a zero tests whether the bit is off, a one tests whether the bit is on and a dot ignores the bit. The following code shows the use of bitmasks to decode the SYSINFO value returned by the PROC COMPARE we ran earlier. Note the use of dots for bits which are to be ignored in each of the tests.

```
/* First trap SYSINFO's value because it will be reset to 0 by any
subsequent procedure or data step */
%let _sysinfo = &sysinfo;

/* Test each bit in turn to see if it's set or not */
data _null_;
    if &_sysinfo = '...............1'b then put 'Bit 1 was set!';
    if &_sysinfo = '..............1.'b then put 'Bit 2 was set!';
    if &_sysinfo = '.............1..'b then put 'Bit 3 was set!';
    if &_sysinfo = '............1...'b then put 'Bit 4 was set!';
    if &_sysinfo = '...........1....'b then put 'Bit 5 was set!';
    if &_sysinfo = '..........1.....'b then put 'Bit 6 was set!';
    if &_sysinfo = '.........1......'b then put 'Bit 7 was set!';
    if &_sysinfo = '........1.......'b then put 'Bit 8 was set!';
    if &_sysinfo = '.......1........'b then put 'Bit 9 was set!';
    if &_sysinfo = '......1.........'b then put 'Bit 10 was set!';
    if &_sysinfo = '.....1..........'b then put 'Bit 11 was set!';
    if &_sysinfo = '....1...........'b then put 'Bit 12 was set!';
    if &_sysinfo = '...1............'b then put 'Bit 13 was set!';
    if &_sysinfo = '..1.............'b then put 'Bit 14 was set!';
    if &_sysinfo = '.1..............'b then put 'Bit 15 was set!';
    if &_sysinfo = '1...............'b then put 'Bit 16 was set!';
run;
```

For our SYSINFO return value of 4113, the following is written to the log:

```
Bit 1 was set!
Bit 5 was set!
Bit 13 was set!
```

Of course we can, and should, make the log messages more meaningful:

```
data _null_;
    if &_sysinfo = '...............1'b then put 'Data set labels differ!';
    /* etc. */
run;
```

Use the band() function

The band() function returns the bitwise logical AND of two arguments. Once again we need to test each bit in turn, as follows:

```
data _null_;
    /* Loop through the 16 bits */
    do bit = 1 to 16;
        /* Convert the binary to decimal */
        code = 2 ** (bit - 1);
        /* Test the current bit */
        if band(&_sysinfo, code) = code then put 'Bit ' bit 'was set!';
    end;
run;
```

Note that the band() function takes decimals as its arguments, not bitmasks.

As before, the following is written to the log:

```
Bit 1 was set!
Bit 5 was set!
Bit 13 was set!
```

The following illustrates how the band() function works. Here we are testing the fifth bit and because the result of the operation is the fifth bit set to one, we know that the fifth bit is set to one in SYSINFO:

SYSINFO return value: argument 1 to band()	0001000000010001
	A N D
Bit 5 (= decimal 16): argument 2 to band()	0000000000010000
	=
Result of bitwise logical AND	0000000000010000

Table 6. Illustration of a bitwise logical AND of a SYSINFO return value and a specific bit

"Bits-to-string" method

This method is a mild variation of the band() function method above, but this time we convert the SYSINFO return value to a string of "bits" using the binary. format first, then we loop through the character positions in the string to test each "bit" in turn:

```
data _null_;
   /* Convert the SYSINFO return value to a string of bits */
   sysinfo = put(&_sysinfo, binary16.);
   put sysinfo=;

   /* Loop through the 16 character positions */
   do bit = 1 to 16;
      if substr(sysinfo, 16-bit+1, 1) = '1' then put 'Bit ' bit 'was set!';
   end;
run;
```

The following is written to the log:

```
sysinfo=0001000000010001
Bit 1 was set!
Bit 5 was set!
Bit 13 was set!
```

A PRACTICAL USE OF BITWISE ENCODING

A bitwise encoding approach can be used to facilitate the validation of macro parameter values where valid values for the parameter are specific combinations of various supported options. Let's say we have a macro which generates a graphic and the macro has a parameter called LEGEND whose value defines if and how the legend for the graphic is to be displayed. Supported options for the parameter, various combinations of which the user can define depending in requirements, are as follows:

DEFAULT
NO (or N)
YES (or Y)
CURVE
BIG_N

It doesn't matter here what these options mean, but was does matter is which option combinations are acceptable to the macro. Similar to how SYSINFO works, we first relate each item to a bit:

Bit	Binary	Option
1	00001	DEFAULT
2	00010	NO (or N)
3	00100	YES (or Y)
4	01000	CURVE
5	10000	BIG_N

Table 7. Bit positions for each supported option in the LEGEND parameter

Then having defined which option combinations are valid we build a list of codes with bits set according to which option is present in each combination:

Binary	Valid option combinations
00001	DEFAULT
00010	NO (or N)
00100	YES (or Y)
01000	CURVE
01100	YES CURVE
10100	YES BIG_N
11000	CURVE BIG_N
11100	YES CURVE BIG_N

Table 8. Binary codes for valid option combinations

Other combinations are invalid and if encountered should cause the macro to abort with a controlled error. But if we are saying, for example, that CURVE BIG_N is valid, then how about BIG_N CURVE? And how many permutations of YES CURVE BIG_N are there? (The answer is 6.) From 8 acceptable option combinations, excluding the Y and N aliases for YES and NO, we find we have a total of 16 potential valid permutations of the combinations to test for. But this doesn't matter! We can see that the code 11000 reflects both CURVE BIG_N and BIG_N CURVE – this is important for us because it is the combination that is valid, not one or other of the two permutations alone. In other words, the order in which the user defines the options is irrelevant for this parameter, and the 8 binary codes we see in Table 8 automatically cover the 16 valid option permutations.

Here is a macro which implements this, using a bitwise encoding approach to validate our LEGEND parameter (the numbers in the comments are cross-references to explanations of the code which follow below):

```
%macro test(legend=) / minoperator;

   %local origlegend parmopts validcodes parmcode pos regid i;

   %* 1 ;
   %let origlegend = &legend;
   %let legend = %upcase(&legend);

   %* 2 ;
   %let legend = %sysfunc(prxchange(s/\bNO\b/N/i,-1,&legend));
   %let legend = %sysfunc(prxchange(s/\bYES\b/Y/i,-1,&legend));

   %* 3 ;
   %let parmopts = BIG_N CURVE Y N DEFAULT;
```

```
    %* 4 ;
    %let validcodes = 00001
                      00010
                      00100
                      01000
                      01100
                      10100
                      11000
                      11100;

    %* 5 ;
    %let parmcode = 00000;

    %* 6 ;
    %do i = 1 %to %sysfunc(countw(&legend));

        %* 7 ;
        %let pos = %sysfunc(findw(&parmopts, %scan(&legend, &i), , es));

        %* 8 ;
        %let regid = %sysfunc(prxparse(s/(.{%eval(&pos - 1)}).(.*)(1)/$1$3$2/));
        %let parmcode = %sysfunc(prxchange(&regid,1,&parmcode.1));
    %end;
    %put Bitwise encoding of LEGEND: &parmcode;

    %* 9 ;
    %if &parmcode in (&validcodes) %then %do;
        %put LEGEND (&origlegend) is valid!;
    %end;
    %else %do;
        %put %str(ERR)OR: LEGEND (&origlegend) is invalid!;
    %end;

%mend test;
```

1. The original value of LEGEND is stored separately for use in the log messages which are issued. The value of LEGEND is also “upcased” to allow the macro to restrict itself to upper case values generally.
2. Here we convert YES and NO in the parameter to their corresponding aliases Y and N. This reduces the number of valid permutations which we need to check for by some margin.
3. Here we assign the list of valid parameter options to PARMOPTS. The position of each item in the list relates to the position of the bit we will use to flag the presence of options both in the various valid option combinations and in the value passed to the macro via the LEGEND parameter.
4. VALIDCODES contains the encoded list of valid option combinations.
5. The next task is to encode the options in the LEGEND parameter. The encoded options will be stored in the variable PARMCODE. First we initialise all bits in PARMCODE to off.
6. Then we loop through the words in LEGEND using…
7. … the `findw`() function to establish the position of the word in PARMOPTS (see 3 above). For example, if Y is present in LEGEND, `findw`() will tell us that Y is the third word in PARMOPTS. Note here the modifiers “e” (return the word number) and “s” (use space characters as the delimiter).
8. For each word in LEGEND we set the corresponding bit in PARMCODE. Taking our example with Y again, we set the third bit because Y is the third word in PARMOPTS. To set the bits in PARMCODE

we are using a regular expression. Although the regular expression may appear a little cryptic at first glance, it's succinct, it does the job and it's easily portable to other option combination testing use cases without amendment. It also adds a surplus bit to PARMCODE should unrecognised words be found in LEGEND, ensuring that such parameter values are deemed invalid. Repeats of valid words are in effect ignored, so if this is a problem for your circumstance code should be added to handle this.

The regular expression which sets the bits uses "search and replace". Let's take a closer look at how this works by dissecting the expression:

```
s/(.{%eval(&pos-1)}).(.*)(1)/$1$3$2/
```

Regex component	Meaning	Capturing group
`s/`	Regex substitution flag	
`(.{%eval(&pos-1)})`	First capturing group: all characters *before* the bit we wish to set	$1
`.`	The bit we wish to set	
`(.*)`	Second capturing group: all characters *after* the bit we wish to set	$2
`(1)`	Third capturing group: matches the dummy "1" added to the end of PARMCODE (see the third argument to `prxmatch`()): `&parmcode.1`	$3
`/$1$3$2/`	The substitution string is made up entirely of capturing groups; note the use of $3 which sets the required bit using the dummy "1"	

Table 9. Regular expression components and their meanings

Here we see the regular expression in action for LEGEND=Y BIG_N. From the macro code above we'll recall that PARMOPTS is set to BIG_N CURVE Y N DEFAULT and that PARMCODE is initialised to 00000:

Regex	**(.{%eval(&pos-1)})**	**.**	**(.*)**	**(1)**	**/$1$3$2/**
PARMCODE before regex substitution for Y (Y is 3rd word in PARMOPTS so POS=3)	**(.{3-1=2}) ► 00** **$1**	**0**	**00** **$2**	**1** (dummy) **$3**	
					$1 $3 $2
PARMCODE after substitution					**0 0 1 0 0**
PARMCODE before regex substitution for BIG_N (BIG_N is 1st word in PARMOPTS so POS=1)	**(.{1-1=0}) ►** [blank] **$1**	**0**	**0100** **$2**	**1** (dummy) **$3**	
					$3 $2
PARMCODE after substitution					**1 0 1 0 0**

Table 10. Regular expression bit substitutions for LEGEND=Y BIG_N

9. Finally we can check to see if the encoded version of LEGEND is present in VALIDCODES and react accordingly.

Thus we have a simple macro which validates the options passed in via the LEGEND parameter and which can easily be extended should additional options become supported in the future or adapted for validating other parameters which have option combinations as their expected values.

Here are some example calls with log output:

Call	Log output
`%test(legend=default big_n);`	`Bitwise encoding of LEGEND: 10001` `ERROR: LEGEND (default big_n) is invalid!`
`%test(legend=big_n yes curve);`	`Bitwise encoding of LEGEND: 11100` `LEGEND (big_n yes curve) is valid!`
`%test(legend=yes badoption);`	`Bitwise encoding of LEGEND: 001001 *` `ERROR: LEGEND (yes badoption) is invalid!`
`%test(legend=no no no);`	`Bitwise encoding of LEGEND: 00010` `LEGEND (no no no) is valid!`
`*` *The additional "bit" here on the right is simply the dummy "1" left behind because the regular expression resulted in no match and because there was no match there was no substitution done. This behaviour is handy for catching invalid options in the parameter value.*	

Table 11. Sample calls to %test and resulting log output

A few words about the BOR() function

No mention of the `band`() function should go without mention of its sister function, `bor`(), which returns the bitwise logical OR of two arguments. This function can be used to set bits and indeed it could have been used in our %test macro instead of the regular expression. The disadvantage for our purposes is that, like the `band`() function, `bor`() takes decimal arguments, which means we would need either to convert our bit strings to decimals for use in the function (= more code), or to do away with the bit strings altogether and rely entirely on decimals. The advantage of bit strings is their clarity within the code, as it is easily seen which bit relates to which parameter option, and this clarity is lost with decimals. The regular expression approach has two advantages as well – not only is the regular expression scalable to any number of valid options without amendment, it also ensures invalid options which may be present in the parameter value are caught.

The following illustrates how the `bor`() function works. Here we are setting the fifth bit to flag the presence of BIG_N in the LEGEND parameter value, the third bit, for Y, having already been set:

Bit 5 (= decimal 16) for the BIG_N in LEGEND=Y BIG_N: argument 1 to bor()	10000
	O R
Bit 3 (= decimal 4) set in PARMCODE for Y where LEGEND=Y BIG_N: argument 2 to bor()	00100
	=
Result of bitwise logical OR (= decimal 20)	10100

Table 12. Illustration of a bitwise logical OR: setting bit 3 for LEGEND=Y BIG_N

As you can probably tell, it doesn't actually matter which way round the arguments are passed to `bor`().

CONCLUSION

Bitwise encoding provides a convenient way to store diverse information within a single numeric code. The SYSINFO return code generated by the COMPARE procedure in SAS contains information about the outcome of a data set comparison and cumulatively flags up to 16 different result types. The MsgBox

function used in VBScript and related languages takes a similar approach to define the buttons and icon for the message box to be displayed. And we have seen a simple practical use of programmer-defined bitwise encoding, namely how the technique can be applied within a macro to validate option combinations passed to the macro as parameter values. The approach offers ease of maintenance, readability and scalability and with a little imagination it can be applied in your own programming to assist with the handling of similar scenarios.

ACKNOWLEDGMENTS

Thanks go to my esteemed colleague Christoph Baumer, likewise of Syneos Health, for his careful review of the paper and constructive feedback.

REFERENCES

Hinson, Joseph; Coughlin, Margaret. 2012. “Deciphering PROC COMPARE Codes: The Use of the bAND Function”. SAS Global Forum 2012.
Available at https://support.sas.com/resources/papers/proceedings12/063-2012.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Rowland Hale
Syneos Health
rowland.hale@syneoshealth.com
syneoshealth.com