Using Bitwise Operators In VB

December 7, 2011 by James H. Byrd

The article on using logical operators prompted a visitor to our site to ask how you use VB's logical operators to *intentionally* perform bitwise operations. (Thanks Cindy!) To demonstrate bitwise concepts, this article shows you how to store an array of Boolean flags within a single numeric variable. It shows you how to use the And, Not, and Or operators to set and test flag values.

If you write VB programs, you probably use bitwise operations already, although you may not have realized you were doing so. For example, if you have ever coded a call to the MsgBox function where you passed something like "vbYesNo + vbQuestion" as the second argument, you were setting flags with a bitwise operation.

The MsgBox function uses a single numeric parameter (the "buttons" parameter) to hold multiple flag settings. One advantage of this approach is that you don't have to pass a laundry list of Boolean flag parameters to the function: you can pack them all into a single "flags" parameter instead. The other advantage is that you can set and test multiple flags with one statement, if you design your flags properly. More on that later.

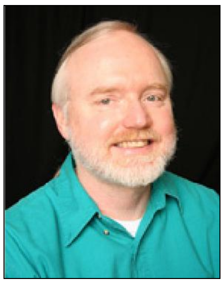
How Computers Store Data in Memory

Before you can delve into the somewhat arcane subject of bitwise operations, you have to know a little about how data is stored in memory. All data is ultimately stored as a series of bits, where each bit has a value of either 1 or 0. Bits are grouped together to form larger data elements. For example, each character in the extended ASCII character set is identified by a pattern of 8 bits grouped together into one byte. Visual Basic uses two bytes (16 bits) to store the data for a variable declared as an Integer, and four bytes (32 bits) for a Long.

Normally you don't think about the memory architecture of a numeric variable. You just store numbers in it and perform math on it. That's the whole point of using a higher-level language like VB. But VB does provide operators that let you twiddle the individual bits of your variables, which gives rise to the term "bitwise" operators. Microsoft's documentation typically calls them "logical" operators, which implies that they perform Boolean evaluations at the expression level, but they really work at the bit level, so don't be fooled (see "Using Logical Operators with Non-Boolean Expressions").

Creating Your Own Bit Patterns

This article uses Integer data types in most of the examples, but the demonstrated principles apply to other data types as well. In general, you should stick to Byte, Integer, and Long when storing bit flags because it is easy to define constants that correspond to the bit patterns you require.



James H. Byrd
Book Publisher
Fiction Author
Software Developer
Technical Writer
Speaker
Vegetarian Cook

Search for Articles

For an example of how you create bit patterns, look at Example 1.

```
Example 1:

00000000 00000000   intI = 0
00000000 00000001   intI = 1
00000000 00000010   intI = 2
00000000 00000100   intI = 4
00000000 00000111   intI = 7
```

The example shows an integer variable named intI. On the right are assignment statements that set intI to various numeric values. The bit pattern corresponding to those values is on the left. The two eight-bit bytes that form the integer variable are separated by a space for readability.

Why does 4 correspond to a bit pattern of "100"? Well, as mentioned earlier, computers must store all data as a pattern of bits. Each bit position represents a specific power of 2. A 1 bit indicates that the corresponding value should be added to the total, and a 0 bit indicates that it should not. This numbering system is called a binary (or Base2) numbering system.

Bit positions are numbered from right to left, starting with 0. Each bit represents 2 raised to the power of the corresponding position number. The leftmost bit of an Integer or Long variable indicates the sign of the number. If the leftmost bit is 1, then the number is negative. Negative numbers don’t follow the same rules as positive numbers. They are stored in a format called "two’s-complement" and you don’t even want to think about that right now. Just stay away from the leftmost bit.

Any number raised to a power of 0 equals 1, so bit 0 in a binary system corresponds to a 1 in the total value. Bit 1 (second from the right) represents 2 to the power of 1, which is 2. If bit 1 is set to 1, then the value 2 is added to the total. In an Integer variable, this pattern continues all the way up to bit 14, which represents 2 the the power of 14, or 16,384. If you turn on all the bits in an Integer except the leftmost bit, the resulting number is the sum of all powers of 2 from 0 to 14, which is 32,767. Now you know why an integer can’t hold a number higher than 32,767.

Now take another look at Example 1. Notice that the bit pattern for 7 is what you get if you add the bit patterns for 1, 2, and 4 together. If you pretend that each bit represents a Boolean flag instead of a power of 2, then 7 represents the situation where the first three flags are all set to True. This concept forms the basis of how you can store multiple Boolean flags in a single numeric variable.

How Bitwise Operators Work

Next you need to know a little about how the bitwise operators work. Most bitwise operators compare two bits and produce a result bit. This article will focus on the And, Or, and Not operators because those are the operators you will need most often. However, keep in mind that VB supplies three other operators that you might find a use for. These operators are Xor (exclusive or), Imp (implication), and Eqv (equivalence). Search for "logical operators" in VB help for more information regarding these three operators.

Take a look at Example 2 to see how the Not operator works.

```
Example 2:

00000000 00000001   intI = 1
11111111 11111110   intI = Not intI
```

The Not operator is a special case because it is a unary operator, which means it acts upon a single operand. It performs logical negation, which flips 1 bits to 0 bits and vice-versa. Example 2 shows an integer variable with the last bit set to 1 and the remaining bits set to 0. When you use the Not operator on the integer variable, the result is an integer with the last bit set to 0 and the remaining bits set to 1.

Now take a look at Example 3 to see how the And operator works.

```
Example 3:

00000000 00000011   intI = 3
00000000 00000110   intJ = 6

00000000 00000010   intK = intI And intJ
```

The And operator compares the corresponding bits of two values. If both bits are 1, the result is 1, otherwise the result is 0. The intI variable has a 1 bit in the last position but intJ does not, so the result (intK) has a 0 in the last position. The second bit from the right is the only bit that is 1 in both integers, so the result has a 1 bit in just that position.

Finally, check out Example 4 to see how the Or operator works.

```
Example 4:

00000000 00000011   intI = 3
00000000 00000110   intJ = 6

00000000 00000111   intK = intI Or intJ
```

The Or operator also compares the corresponding bits of two values. If either bit is 1, the result is 1. If both bits are 0, the result is 0. Since one or the other of the two values has a 1 in the last three positions, the result contains a 1 in all of the last three positions.

Now that you are armed with a rudimentary knowledge of bit patterns and bitwise operators, you are ready to see an example that puts that knowledge to work.

Creating a Bit Flag Variable

To create and work with your own bit flag variable, you first need to decide on what type of variable you want to use. That decision depends upon how many bit flags you require. If you have 8 or fewer flags, you can use a Byte variable because the Byte data type consists of 8 bits. If you have 15 or fewer flags, you can use an Integer variable, which has 16 bits (remember to leave the sign bit alone). If you have more than 15 flags, you should use a Long, which is 32 bits and allows for 31 flags. Most flag variables are either Integer or Long because those data types are highly portable to other data and development environments.

The next thing to do is decide upon the layout of your bit flags. For a practical example, this article uses the Shift argument in the KeyDown event, which happens to be an integer. The Shift argument has three flags, each of which reflects some aspect of the shifted state of the keyboard. The layout of the flags is shown in Example 5.

```
Example 5:

Bit 0: Shift Key: 1=Shift key is held down, 0=Shift key is up
Bit 1: Ctrl Key: 1=Ctrl key is held down, 0=Ctrl key is up
Bit 2: Alt Key: 1=Alt key is held down, 0=Alt key is up
 (remaining bits unused)
```

Ideally, each bit is the answer to a single yes/no question, although that is not always the case. The MsgBox "button" argument, for instance, uses multiple bits to represent a single condition that has more than just two states. Also, it is easiest to work with bit flags that are completely independent of one another: that is, the value of one flag should not depend upon the value of another flag. Even if your flags are somewhat interdependent, you can still use bit flags, but you have to be more careful about how you set the flags.

Once you have a bit flag layout, you can define some constants to represent the required bit patterns. Example 6 shows the constants for the Shift argument along with their corresponding bit patterns:

```
Example 6:

00000000 00000001   Const vbShiftMask = 1
00000000 00000010   Const vbCtrlMask = 2
00000000 00000100   Const vbAltMask = 4
```

Setting Bit Flags to True

You can use the constants you define and the Or operator to set individual bits in your flag variable to True. In the Shift argument example, Visual Basic does this for you when it raises the KeyDown event. But for now, pretend that you need to set these flags yourself.

Assume, for example, that you need to turn on the flag corresponding to the Ctrl key. Example 7 shows how you would accomplish that:

```
Example 7:

00000000 00000010   Const vbCtrlMask = 2
00000000 00000000   Dim Shift As Integer

00000000 00000010   Shift = (Shift Or vbCtrlMask)
```

In the above example, you could have simply assigned vbCtrlMask to Shift and achieved the same results, but that approach would wipe out any other flag settings that may have already existed in the Shift variable. The next example demonstrates this concept:

```
Example 8:

00000000 00000010   Const vbCtrlMask = 2
00000000 00000100   Const vbAltMask = 4

00000000 00000010   Shift = vbCtrlMask
00000000 00000110   Shift = (Shift Or vbAltMask)
```

The first assignment statement simply copies the bit pattern for vbCtrlMask to the Shift variable. The second assignment statement uses the Or operator to copy the 1 bits from vbAltMask to the Shift variable without disturbing the 1 bits that were already in Shift.

You can, of course, perform more than one Or operation with a single statement. If you need to set several flags at the same time, you can just Or the constants together:

```
Example 9:

00000000 00000111   Shift = (vbShiftMask Or vbCtrlMask Or vbAltMask)

Note that you would get the same result if you added the three flags together like this:

Example 10:

00000000 00000111   Shift = (vbShiftMask + vbCtrlMask + vbAltMask)
```

In fact, most programmers use this approach when passing the buttons argument to the MsgBox function. But be careful! Addition only works properly when you initialize your flag variable. If you add a flag constant to a variable that already has that flag set, the 1 bit will be cleared and carried to the next position of the pattern, which is not what you want at all:

```
Example 11:

00000000 00000010   Shift = vbCtrlMask
00000000 00000100   Shift = (Shift + vbCtrlMask)
```

Doh! By adding vbCtrlMask to Shift, you accidentally turned on the Alt key flag, and the Ctrl flag isn’t set at all now. Your best bet is to always use the Or operator if you are performing a bitwise operation, because it will always work reliably.

Examples that show you how to set bit flags to False come later in the article. For reasons you will soon understand, that discussion makes more sense after you know how to read bit flags.

Reading Bit Flags

The key to reading bit flags is the And operator. You can use the And operator with your constants to determine which bit flags are set in your flag variable. This operation is frequently called "masking," which accounts for the naming of the VB constants for the Shift argument. The idea behind a mask is that it uses 0 bits to mask out all of the positions you don't care about and 1 bits for the positions you do care about. Here's an example:

```
Example 12:

00000000 00000010   Const vbCtrlMask = 2
00000000 00000111   Shift = (vbShiftMask Or vbCtrlMask Or vbAltMask)

00000000 00000010   If (Shift And vbCtrlMask) > 0 Then
```

Remember that the And operator compares two bit patterns and produces a result with a 1 bit only where both patterns have a 1 bit. The If statement in Example 12 is checking the Shift variable to see if the Ctrl key flag is set.

So can you check more than one flag at a time? Sure! You can do this a couple of ways. One way is to define a constant that has both flags turned on, and use it with the And operator. The other approach is to use the Or operator to combine the two constants on-the-fly and then use the result with the And operator. The example below demonstrates both techniques:

```
Example 13:

00000000 00000001   Const vbShiftMask = 1
00000000 00000010   Const vbCtrlMask = 2
00000000 00000100   Const vbAltMask = 4
00000000 00000011   Const conShiftOrCtrlMask = 3
00000000 00000000   Dim Shift As Integer
00000000 00000000   Dim Test As Integer

00000000 00000111   Shift = vbShiftMask Or vbCtrlMask Or vbAltMask
00000000 00000011   Test = Shift And conShiftOrCtrlMask
00000000 00000011   Test = Shift And (vbShiftMask Or vbCtrlMask)
```

Setting Bit Flags to False

The earlier examples showed you how to turn a bit flag on, but what if you want to turn a flag off? That is where the Not operator comes in. Take a look at what happens when you use the Not operator on the Ctrl key mask:

```
Example 14:

00000000 00000010   Test = vbCtrlMask
11111111 11111101   Test = Not vbCtrlMask
```

Using the Not operator on any mask essentially gives you a negative image of that mask. If you use this negative image mask with the And operator, the result will pass through every 1 bit *except* the bits in the original mask. The following example demonstrates this concept:

```
Example 15:

00000000 00000011   Test = (vbShiftMask Or vbCtrlMask)
11111111 11111101   NotMask = Not vbCtrlMask
00000000 00000001   Test = (Test And NotMask)
```

The NotMask variable in Example 15 masks out just the Ctrl key bit, so every other bit position from the Test variable passes through to the result. Consequently, this example turns off the Ctrl key bit without disturbing any of the other bits.

When to Avoid Bit Flags

There are definitely circumstances where you should avoid bit flags. The most obvious reason is that using them can become a bit complicated. The potential for logic errors is high unless you (and your fellow developers) have a firm grasp on how bitwise operators function.

Another time to avoid bit flags is when you are storing the information in a database. Most database back-ends do not support bitwise operations in SQL statements, so you can't test individual flag settings to filter your result set. Plus, ODBC (version 3 anyway) doesn't support any scalar functions that provide this capability, regardless of what the back end can do.

Bit Flags and VB IntelliSense

One annoying aspect of creating constants for your bit patterns is that you then have to remember the constant names when you start writing code. There is an easy way to solve that problem. You may have noticed that VB prompts you with the list of available constants when you reach the buttons argument of the MsgBox function (unless you have the Auto List Members option turned off). You can do the same thing for your own functions and variables by creating an enumeration with the Enum statement.

For example, this is what the Enum statement would look like for the Shift argument constants:

```
Example 16:

Enum ShiftConstants
   vbShiftMask = 1
   vbCtrlMask = 2
   vbAltMask = 4
End Enum
```

The trick is to then declare your variables and parameters using the Enum name:

```
Example 17:

Dim Test As ShiftConstants
Test = vbShiftMask
```

As soon as you type the equal sign in the assignment statement shown above, VB drops down the list of ShiftConstants for you to choose from. Microsoft calls this feature IntelliSense, and it is great for productivity.

If you look in the object browser, you will find that the ShiftConstants enumeration really does exist and it contains the constants presented here. So why didn't Microsoft declare the Shift argument in the KeyDown event as a ShiftConstants data type? Because the Shift argument is an Integer, and the underlying data type for an enumeration is Long.

That is the main down side to using an enumeration for your constants: your flag variables must be of type Long, even if you don't need all those bits. However, the IntelliSense feature is probably worth the trade-off.

Conclusion

Some of you may have moved from other languages where bitwise operations are more common and you just need an understanding of what tools VB offers in this regard. Hopefully, this article helped you out.

Even if you don't create and use your own bit flag variables, you will undoubtedly encounter circumstances where you must deal with them. The Windows API, for instance is rife with them. Learning how to properly use bitwise operators will help you deal with these situations confidently.

If you would like to use bit flags in your application, but you are concerned about the maintainability of the resulting code, then consider printing out the Bit Flag

Cheat Sheet at the end of this article for future reference by you and your development team.

Bit Flag Cheat Sheet

```
Sample Declarations:

00000000 00000001   Const conFlagOneTrue As Integer = 1
00000000 00000010   Const conFlagTwoTrue As Integer = 2
00000000 00000000   Dim intFlags As Integer

Initialize both flags to true:

   intFlags = conFlagOneTrue Or conFlagTwoTrue

Set flag one to False without changing flag two:

   intFlags = intFlags And (Not conFlagOneTrue)

Set flag one back to True without changing flag two:

   intFlags = intFlags Or conFlagOneTrue

Test flag one for True:

   If (intFlags And conFlagOneTrue) > 0 Then

Test flag one or flag two for True:

   If (intFlags And (conFlagOneTrue Or conFlagTwoTrue)) > 0 Then
```

Filed Under: MS Visual Basic, Nerdy Musings, Windows Development Tagged With: Bitwise Operations, COM Development, Visual Basic 6